AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI
INSTYTUT INFORMATYKI

Projekt dyplomowy

A system for supporting effective usage of power produced by a photovoltaic power plant

System wspomagający efektywne wykorzystanie energii produkowanej przez farmę fotowoltaiczną

Autor: Jakub Janicki, Adam Przywieczerski
Kierunek studiów: Informatyka
Opiekun pracy: dr inż. Sławomir Zieliński

Kraków, 2023

Contents

1. Project goals and vision

1.1. Problem specification

Microgrid architecture is becoming more and more popular in energy production, replacing traditional centralized utility grids [19]. A microgrid is a group of interconnected energy users and distributed energy resources, that can work independently of the main grid. It often uses renewable energy as the source of power [19]. One of the advantages that come with this approach is the possibility to avoid the costs of purchase and distribution of electricity, through production per customers' needs. Nevertheless, it is difficult to meet customers' demands and keep sufficient production levels of energy, since they are highly dependent on weather conditions [25]. One idea is to expand local powerhouses with additional storage to gather energy for when the conditions are unfavorable. This solution tackles the problem of differences in energy production and consumption but only in short time periods. We can only store small amounts of energy, so managing it over months with different production levels is impossible. To add to that energy storage systems are not profitable nowadays [18].

The problem of small storage capacity can be solved by keeping the energy in the public grid. The agreement conditions vary by the contractor, farm power and customer. Private farmers whose production levels were lower than 10kW were in the most advantageous situation. Up to April 2022 they could store the produced energy in the public grid and had the ability to regain 80% of it each year for free. Now new laws have been introduced and farmers cannot store the energy, but have to "sell it" (the amount will be deducted from taxes) at wholesale price to the public grid. If they wanted to regain some of it they would have to pay a retail price for it. Arrangements signed right before the change will still be treated accordingly to the old laws, but every new one will not. The above mentioned new system, called net-billing [24], significantly reduced profitability [27]. That is why we decided to help mitigate the downsides of storing energy and find another way to use the produced energy that a farmer might not need at the moment. Instead of giving the energy away for cheap he could use it to power some household appliances or a cryptocurrency miner.

Our goal is to develop a system for smart energy usage that we hope will keep incentivizing people to invest in photovoltaic installations, since some of them may have been scared off by the recent law changes. The system based on the data provided by measuring devices and the set of rules defined by the user will decide on how to divide the surplus of energy between consumer IoT devices. We have chosen photovoltaic installations as our power source, because they are becoming more and more common in Poland. [15] This kind of installation is the most popular backyard power station in the country.

1.2. Existing solutions

So far such solutions weren't necessary, but since the introduction of the new laws a dire need has arisen for an alternative. Apart from small private companies offering simple half-solutions of creating networks of IoT devices, there aren't many companies on the Polish market that provide such solutions. One of them is *Hewalex* - they offer connecting the farm with heat pumps or water heaters. Besides a limited choice of types of devices, the company also only supports appliances manufactured by them.

1.3. Approach to the problem

Our system has two main features, of which only the first one can be found in other systems:

- checking the ratio of energy being produced and consumed in the local network.
- controlling the use of the surplus of energy on a variety of devices, not necessarily from the same manufacturer.

The second feature is specific to our idea, as most of the already existing solutions cover only the management of devices of the same type or coming from one producer.

As mentioned before we cannot control production levels. Instead we can manage energy consumption. We can handle it by deciding when to use our own energy-consuming devices. This way, we can consume energy with household appliances or transform it into different types of energy. E.g. thermal energy is easier and more common to store, so there is usually no need to provide extra room for storage, eliminating additional costs. In addition in today's world any energy surplus could also be used to power a cryptocurrency miner, making it much more environment friendly.

Since there are many ways to use renewable energy, there is also a wide variety of consumer devices that we can use in our system [26]. Every machine has its own energy demand, which we have to take into consideration while modulating total consumption. Otherwise, we could find ourselves in a situation where more energy is being consumed than produced. Not only energy usage is volatile, but also energy production. It is impossible to predict both parameters, so real-time tracking is essential. This requirement makes manual energy management nearly impossible.

That is why we decided to develop a software system for smart energy management. Its sole purpose is to control a local network of IoT devices (each of them chosen by the client) so that the energy surplus is not wasted. The system is divided into modules, each with a corresponding set of tasks and responsibilities. It tackles all the previously mentioned problems. It is constantly tracking energy usage and production. On this basis, it automatically manages which devices to turn on and off and what task each of them should perform. One of its main features is the ability to control various devices each with a different range of capabilities. The system also provides a visual representation of undergoing processes and data for the customer. Thanks to that the user is able to track the statistics based on the recent periods and potentially tweak the settings of the system to better cover his needs.

We decided to divide the system into 4 major modules that communicate with each other and make decisions based on the shared data:

- Measuring
- Device controller
- Energy usage policy
- Monitoring and managing

1.3.1. Measuring

This is the module responsible for supplying the necessary information about energy production and consumption to the whole system. It reads all of its required information from the electrical hook up point. The supplied readings are the most important data in the entire system - its behaviour is dependent on them. The module additionally writes down the data in the database to store for future use.

1.3.2. Device controller

Through the device controller the system communicates with each connected consumer device. We want our project to be capable of handling any device that fits the given interface. In order to accomplish that the consumer will have to do some work himself. Since we cannot predict what kind of device he will want to use, we have to create a way to exchange information between the system and the newly added device.

For each unique device there will have to be written a dedicated controller with which the Device controller module will communicate. The reason why the support for different devices is such an important feature in our system is that we want our product to be used by all types of customers. Each of them may have various needs and ideas on how they want to use their excess energy. This way we can make the system as universal and attract as many users as we possibly can.

1.3.3. Energy usage policy

Along with the gathered data this is the most important part of the system. It is in here that the rules of energy consumption based on the current available resources are written down. It's going to be the client's job to specify how he wants to use the surplus. The whole module has to be a spacious ruleset covering every possible scenario that may occur during the system's runtime.

When adding a new device to the system the client will have to deliver a complete list of functions that its controller implements and arrange them in such a way, that the system knows which functions to prioritize over others. Each device should also have its own priority. Based on that the system will be able to assess which device to pick first if there is extra energy to use.

The module is also querying the database to check which devices are on and what each of them is doing. This way we won't run into a problem of delegating a task to a device that is already performing it. The gathered data could also be used to show the activity of every device in the system. Based on that the user will be able to see which devices are used and how often. Potential inaccuracies in planning may reveal themselves this way if it turns out that a device is almost never used.

1.3.4. Monitoring and managing

This is the module that connects all of the other ones, making them work as a single system. It gets its necessary data from the Measuring module and the Energy usage module and based on the gathered information it commands a Device controller module to perform particular tasks. It also updates the state of all the devices in the database in case any of them changes. It is also

through this module that the user will be able to make changes in the system, as well as view processed data.

1.3.5. Database

An extra module in the system is the database in which we keep all of the measurements done by the system. Apart from that we are also storing information about connected devices and their status. The database will also keep historic records which could be used to exhibit the system's behaviour in particular timespans. All of the other modules have access to the database, be it reading or/and writing.

1.4. Collaboration diagram

In the diagram (fig. 1) we show the overall vision of the system's modules and their dependencies on each other.

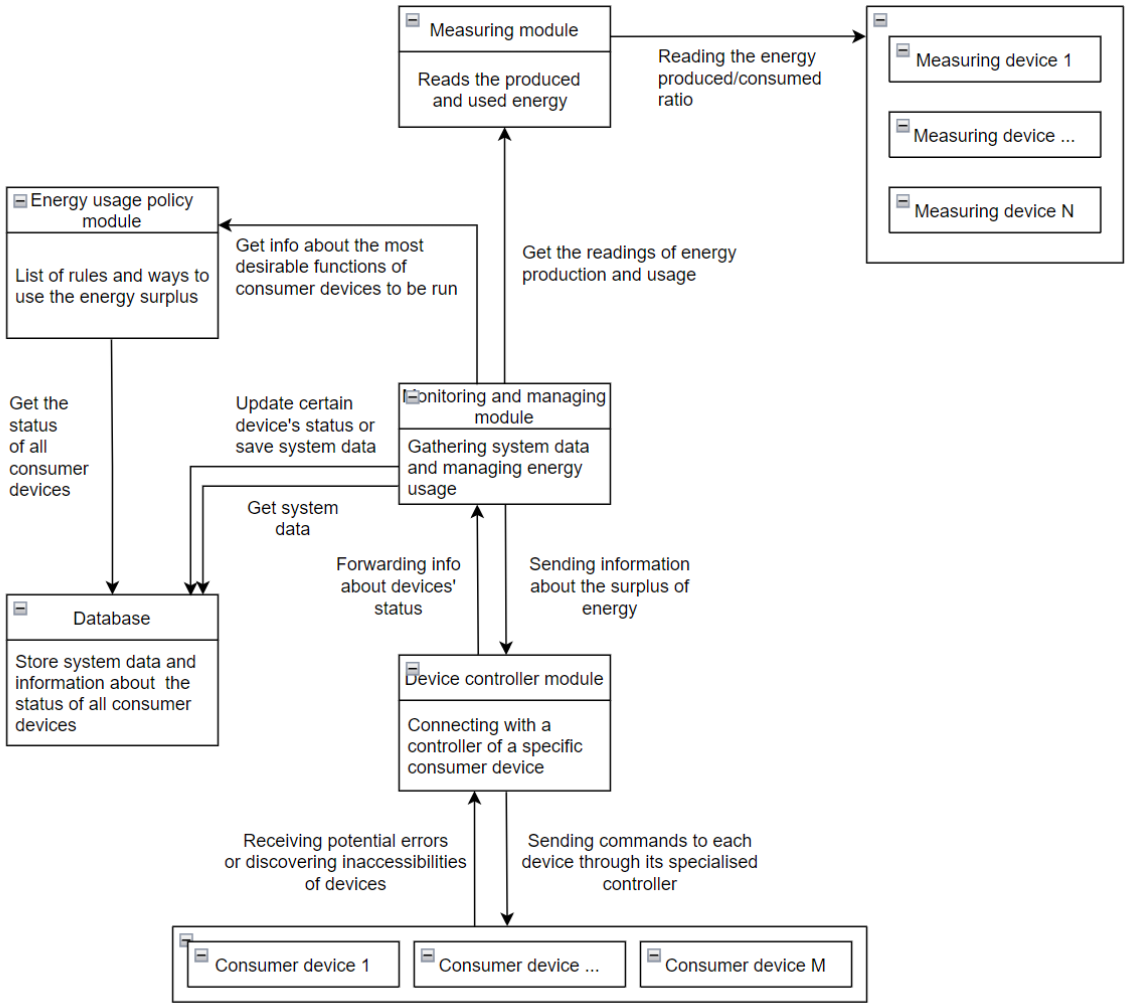

Figure 1: Collaboration diagram

The monitoring and managing module is at the heart of our system connecting all of the other modules and supplying them with necessary information. The measuring module communicates with measuring devices to get the latest readings of power production. The device controller module is responsible for sending requests to consumer devices and forwarding information about their status to the rest of the system. The energy usage policy module selects which devices the system should run based on the provided data.

1.5. Risks to be aware of

In order to honestly assess our project we should also look into possible shortcomings, inaccuracies and errors that we can make along the way. Here are the ones to be especially careful about:

- We had to find a reliable way of determining when our system should act (e.g. a cloud passing by and temporarily lowering energy production should not change the state of the entire system)
- Energy fluctuation caused by consumer devices should be omitted as much as possible. We do not want our system to send conflicting orders to the same device in a short span. A possible solution would be to set a minimum time that needs to pass before the system can act on the same device.
- The difficulty of adding a new device to the system by the customer should not be too high. We have to make sure to have a rather simple and clearly defined way of doing such a task.
- The application to control the system should be visually neatly planned out and easily navigable by the common user. The capabilities of the system should be simply accessible, no matter how difficult they may be.
- Some IoT devices might not expose public API documentation. This can make it impossible to connect the device and consequently to manage it with our system.
- Our system's work heavily depends on the connected devices' reliability.

1.6. Final thoughts

With good planning and enough time we believe we will be able to accomplish all of the above mentioned goals and create a well functioning system. On top of that we should also be able to test our project's performance on an actual photovoltaic farm. Even though the farm might not use all of the system's features it will still provide valuable data and help us ensure the project's integrity over a longer period of time.

We hope that our system will encourage people to keep investing in household renewable resources powerplants while also helping them save some money. Besides costs reduction, photovoltaic installations can also act as a backup energy source in case a failure in the global grid occurs. They also cut out or at least limit the use of non-renewable resources which are now the main energy source in Poland. What follows is the reduction of air pollution and carbon dioxide emission, which is required from all European Union member countries [28]. On top of that government organisations financially support photovoltaic installations due to the "Polish Energy Policy until 2040" [22] of which aim is to increase energy efficiency and minimize the negative impact of energy on the natural environment.

As we have shown, photovoltaic installations are a safe investment and our product will make them even more profitable. Nevertheless, our system will need to be customized for every client, so it is crucial to include its cost in the profitability analysis. First-time customers will have to cover the costs of customization based on their requirements. Fortunately some of those costs can be lowered thanks to a government funding called "Mój Prąd 4.0" [23] which includes smart systems for energy usage in households.

Currently there isn't a similar system on the Polish market and since we believe it will prove to be useful, we decided to fill the niche by developing it ourselves.

2. System functionality

This chapter shows the functionality range of our system and the differences between actors that inhabit it. The main objective is to show the decisions made about the project and the reasoning behind them.

2.1. Types of actors

In our system we will distinguish two types of actors, each with its own set of capabilities:

- Normal user
- Privileged user (Admin)

2.1.1. Normal user

A fairly passive actor with not a lot of control over the system and its internal functionalities. His access to the system's "write" and "execute" functions is non-existent. The main purpose of the actor is to monitor the automated distribution of resources through the mobile app and possibly use the gathered data for further investigation.

In a real world setting the actor could be a child keen on observing the undergoing processes in a household installation, but not trusted enough to have access to higher privileges or a company worker delegated for this exact job. Most of the users in every instance of our system should be of this type as even an unauthorized access would have no chance of compromising the workflow of the system.

2.1.2. Privileged user (Admin)

An actor with the highest privilege that has complete control over the running system. To this group we can count in all of the adults in the household or at the very least the person responsible for the whole setup. In an industrial setting this can be the owner of the company or a higher-up responsible for overseeing the operation.

Every first user in the system's database must be an admin - without one there would be no point in using our product as we do not want to give normal users too much acting power. We expect that each farm will only have one privileged user and prefer to keep it that way for the foreseeable future. An admin holds all the power in the system, so to prevent any complications and dangers that come with introducing multiple such actors, we limit it.

The admin shares all of the "read" capabilities of a normal user and extends them beyond that, having the ability to affect the system to his liking. He is also responsible for the creation and management of all other users in the system.

2.2. Specific functionalities

We decided to describe our project's main functionalities and desired behaviors in the form of User Stories, based on the **MoSCoW** prioritization method [14], which divides project's expectations into four groups - **Must**, **Should**, **Could** and **Would/Won't Have**. It allows to primarily focus on the more important parts of the project while slowly making way towards the lesser needed functionalities. Each of the categories has one defining characteristic.

- **Must** - critical features absolutely necessary for the successful completion of the project. All of them must be implemented in the release version of the product.
- **Should** - functionalities that should come with the product. They are not vital to the project's completion status but are expected to be included.
- **Could** - requirements that improve the overall user experience, but are not necessary.
- **Would/Won't have** - a set of functionalities of the lowest priority. Depending on the variant it's highly unlikely that they will be included in the current version of the product (**Would**) or they certainly won't be, but could be in the future (**Won't have**). We've decided to use **Would** as our final category.

Below is the list of our system's requirements. The admin shares all of the capabilities of a normal user. If a feature is specifically intended for the admin it is noted in the Story.

2.2.1. Must

- As a system owner I want the system to efficiently control devices based on the defined policies
- As a system owner I want the ability to register an admin account and log in as such into the web and mobile applications
- As an admin I want to be able to create normal user accounts
- As a user I want to be able to log in to the mobile application
- As an admin I want the ability to create a farm to which I can later assign devices
- As an admin I want to have the ability to perform CRUD operations on the system's components such as measuring and consumer devices
- As an admin I want to have the ability to define multiple tasks for a device depending on its complexity
- As an admin I want to be able to organize devices' tasks in groups depending on the level of energy being produced
- As a user I want to have access to real time and historical measurements gathered by devices
- As a user I want to know the status of all connected devices

2.2.2. Should

- As an admin I want to have control over the algorithm through editable parameters and an on/off switch
- As an admin I want to be able to customize information about a device such as name or additional labels
- As a user I want to be able to view devices' activity history
- As a user I want the available data to be shown in the app in the form of graphs

2.2.3. Could

- As an admin I want to be able to choose between different methods of distributing available resources (e.g. different algorithms)

2.2.4. Would

- As an admin I want to be able to give each device a distinct photo representing it in the app

2.3. Functionalities in detail

The entire product is divided into three parts - the server side, which is responsible for the system's operation, the web application through which the admin has the ability to modify the devices' network and change system policies and the mobile app which gives easy access to check on the running processes and provides data about them in a digestible way. All parts cooperate in creating the workflow of the whole product.

2.3.1. Login

Before accessing the system each user must first have an account registered in the database. Every session must start with logging in to the app - be it the mobile app for a quick checkup on the network or the web application to make changes in the system. Both the admin and a normal user have the same view of the mobile application as it serves only for monitoring. The admin's privileges also allow him to access the web application and make changes in the internal workings of the product.

For a normal user to log in, he first needs an account to be created by the system admin. Only then can he access the mobile app.

2.3.2. Create a farm instance

A farm is what we define as a single instance of our bare product - without access to data and control over the processes. It consists of measuring and consumer devices and an algorithm that binds the two together.

2.3.3. Add consumer device

One of the most important functionalities without which the whole project would not work. It allows the admin to introduce a new consumer device into the system, bringing more complexity with it. In the process of adding a device the user will be able to label the tasks that it can perform and define parameters required for each of them.

On top of that every appliance must have a name and a logical placement in the devices' network so it can be reached by the system. For that to happen the user must also provide an IP address for each of his devices and a farm to which it belongs.

2.3.4. Remove consumer device

The exact opposite of the previously mentioned function. If a device is no longer connected to the network or the admin does not want to acknowledge it in the logical layer he can simply

remove it from the app and consequently from the entire system. After deleting a device every piece of data about it will be removed from the system along with all of the custom settings for it.

This function should not be taken lightly as the only way to revert it is to add the device back to the network through the app or to manually insert it into the database along with all of the necessary data.

2.3.5. Add measuring device

The function allows for adding of the measuring devices to the system's grid. The measuring devices along with consumer ones are the most crucial part of the project. The former supply the system with the knowledge about the available surplus of energy, the latter create an output for that energy to be used.

As is the case with adding of consumer devices the user must provide a name, an IP address and select a farm for each appliance for it to be recognizable by him and the system. There is no need for any extra data as measuring devices are supposed to be running all the time and their only role is to gather readings about the difference in energy production and consumption and share it to the system.

2.3.6. Remove measuring device

Removing a measuring device is not as serious of an operation in the difficulty of reverting the change, but absolutely detrimental to the product's workflow. It strips the system from the information about the available energy for distribution hindering the entire operation in the process. Because of that it should be performed with utmost care and caution.

2.3.7. Edit device's list of tasks

As we expect users to connect different types of IoT devices to our product, each with its own unique set of capabilities, we must provide a way to include them in our system. That is why every consumer appliance will be connected with its own list of user defined functions. Each task must have a label, an energy consumption level and possible parameters so the system knows how resource demanding it is and what additional information might be needed. If the user forgets to add a device's functionality, wants to change its details or no longer desires one to be performed he can edit the list to his liking. The device simply existing in the system with its functionalities does not mean it will be taken into account by the ongoing algorithm - it has to be manually configured beforehand.

2.3.8. Create energy ranges for devices' activities

When it comes to the energy being produced by the farm the user will know his system best. He will be aware of the minimal and maximal values, how often they occur and for how long they stay at certain levels. Because of that we've decided to give the users the ability to define subsequent energy ranges to which they will be later adding devices with their chosen functionalities to be run. The ranges' thresholds, the devices and their selected functions will be adjustable, so the system, relying on users' predictions of weather conditions, can stay as efficient as possible at all times.

2.3.9. React to manual changes on devices

The system is capable of handling basic manual changes on devices. If a user turns a device on or off without the use of our product, the system will not overturn his decision and will instead ignore the device for a specified period of time. After the time passes the system will go back to managing the device.

2.3.10. Overview of consumer devices

Through the app the user has constant access to information about the status of each consumer device in the system, including a chart showcasing its activity.

2.3.11. View production levels

Each measuring device is constantly gathering information about the energy being produced in the different parts of the grid - as is the case in bigger photovoltaic installations. The user has the ability to access this data to keep an eye on the process and potentially detect abnormalities. The real time data along with the historical data also helps to outline predictions and trends in the production.

2.4. Nonfunctional requirements

Besides the different capabilities of the system we still have certain expectations as to how reliant and efficient it will be. To ensure that we are happy with the results we highlighted a couple of resolutions to focus on, so that every decision in the product's development brings us closer to achieving them.

2.4.1. Algorithm efficiency

The algorithm must be highly responsive and not make logical mistakes, even minor ones, in its inner workings as it is the core of our system. It has to be easily understandable by the user and simple to edit.

2.4.2. Ease of device diversification

The system must be capable of managing different types of devices. The communication between the two must be unified across the system, but the commands sent to certain devices have to be done with the use of specifically prepared controllers. The addition of a new type of device to the system along with such a controller must be simple from a programming standpoint.

2.4.3. Ease of use

Creating an entire network of different types of devices and then linking them together might seem like a huge headache in the eyes of many. That is why the initial setup of the system has to be as clear and easy as possible. If a user makes a mistake during the configuration it must be easy to catch and fix.

2.5. Summary

In the above chapter we described the expectations for our system's functionalities, the different types of users and their scope of permissions and responsibilities.

3. Chosen implementation aspects

In this chapter we will dissect our product, focusing on the importance of different of its components, methods of implementation and the technology used in each of them. We'll discuss why certain approaches were made, what difficulties we encountered along the way and how we solved them.

3.1. System architecture

According to the requirements our project consists of the following components:

- Web application - for admin use only, allows for hands-on system management.
- Mobile application - for normal users, provides monitoring data about the system's state.
- Management service - responsible for automated management of devices, collecting data and handling application requests.
- Authentication service - responsible for authenticating and managing users.
- Computing service - performs difficult calculations
- Database - stores data about the system's behaviour
- Simulation tool - for developers' local use. Created in order to simulate and track system's behaviour.

All of the above elements but the last take part in creating a coherently working system. As an additional tool, mainly for testing purposes, we also provided the Simulation tool, which might prove useful when making changes to the system.

3.2. Technology in detail

While choosing the technology needed for our project we decided on two main focus points. Where it was possible we would use the technology we already knew - thinking about the amount of work ahead of us, we did not want to make the entire development process even harder for ourselves.

When it came to areas where both of us had little to no experience we would opt for the most common solutions - including languages, frameworks and services.

The backend part of the system is mostly written in Java using the Spring framework with a sprinkle of Python and bash scripts for some smaller tasks.

The frontend applications are written in JavaScript with the additional use of React (web application) and React Native (mobile application).

Most services are running in the cloud. For the database we used a cloud version of the database provided by MongoDB. The parts of the system that must be running at all times are deployed on the Heroku service and managed there. The system's dependencies and mainly used technology can be seen on the Fig. 2.

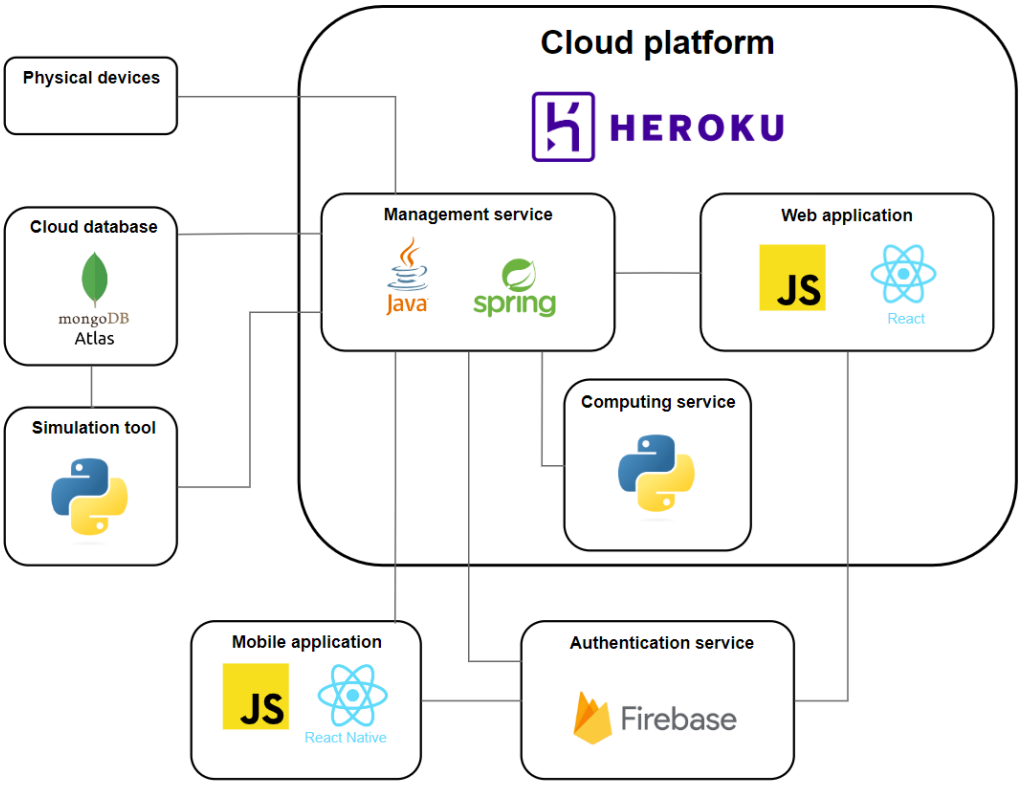

Figure 2: System technological diagram

3.2.1. Web application

The web application was the last element of our product to be implemented. In the initial stages of our works we had to decide how users would interact with our system. After some thought we came to a conclusion that data availability was our top priority at that moment and chose to implement the mobile application. In the later stages when the time came to give the admin the ability to edit the system's devices and behavior we started shifting our focus more to the web application.

Since time was rather scarce we have decided to use a template [11] for the overall application view and customize it using **JavaScript** with **React**. The usage of the UI template allowed us to quicken the app development, since styling - not an important part of our system - has been already done. Thus we could focus more on the logical behaviour of the app. We also used React Hook Form library [7] in order to limit the number of renderers and add validation to the user's input. As a state management tool we have chosen the React Redux library [9], thanks to which we have global access to the app's state.

Communication with the Management service was done through the use of a **REST API**. The admin can send all kinds of requests - depending on the functionality.

GET requests deliver the current state of the system. Despite the lack of interest in detailed data the admin still has to know which devices are connected and what critical information they hold.

PUT / POST requests are used for adding and updating resources in the system, which includes:

- farm instances
- measuring devices
- consumer devices
- users

DEL requests allow for resource deletion. Each such request comes with serious consequences. If we decide to delete a farm instance all of the devices assigned to it become idle. They do not disappear from the system, but until further notice they do not perform any actions.
If we delete all of the measuring devices from a farm, the same situation occurs.
A consumer device deletion has its consequences when it comes to the system's way of distributing resources. Every mention of the device also has to be deleted from the pool, so the admin has to be aware of an ongoing strategy during each such action to update it if necessary.

3.2.2. Mobile application

The mobile application is written fully in **JavaScript** with the use of the **React Native** framework. There were other equally popular alternatives such as .Net with Xamarin or Dart with Flutter, but having some experience in JavaScript and React convinced us to choose a familiar technology.

As it turned out it wasn't a bad choice. React Native has an enormous community, an abundance of tutorials, custom solutions to many problems and - what's most important - a sizeable gallery of libraries all accessible through the **npm** tool.

On top of that it is also a cross-platform framework - meaning that one version of a codebase can be used on both **Android** and **IOS**. Of course some differences in the code must be taken into account when dealing with platform specific components or behaviors. Nonetheless implementing the necessary considerations is far less time consuming than creating the same application twice using different tools each time.

In our product we have only covered the use of the app on an Android phone, since neither of us had access to an iPhone and working with a physical device instead of an emulator was much more handy. In the future, if the situation arises, it would be easy to configure the app to handle IOS based devices as well.

The mobile app has two main functions that it needs to perform:

- fetching data through the **Management service**
- charting data

In theory the first task was going to be quite easy as long as all of the necessary endpoints were accessible. In reality the asynchronous nature of JavaScript along with React Native caused some difficulties resulting in the rendering of certain views in the app before all data was loaded. The used components were also at times 'Pure' [12], which made it necessary for us to come

up with ways to force them to update. The process of acquiring data is done through the use of a React Native *fetch* function that allows for calls to a REST API. We did not plan for a two-way communication between the app and the server, so a polling mechanism covers all of our needs.

Data charting was done with the use of the ***react-native-responsive-linechart*** library [8]. Having tested a couple of different libraries we have decided on this one, because it allowed for an interactive use of charts. The limited space on the smartphone screen often calls for compromises and since we wanted to keep as many characteristics of the presented data as possible we needed a way to display it in its entirety and detail. The used library supports scrollable charts with easily editable domains and labels on both axes. Despite being more robust than the other considered libraries it also allows for fairly deep chart customization.

3.2.3. Management service

The management service is the core element of our system. Not only does it manage all of the devices and provides user applications with data, but also allows for extending its own use through a specific architectural design choice allowing for addition of new devices with their own set of unique features. This quality is what makes our system truly stand out and makes it widely more applicable than if we were to focus on only certain types of appliances or models.

Object oriented

The service has been designed in the **Object Oriented Programming** paradigm in **Java** with the use of the **Spring** framework, which provides a dependency injection mechanism [1]. It heavily relies on communicating with different devices (or rather their drivers), so the usage of interfaces was inevitable. The devices often being similar to each other also allow for code reusability thanks to inheritance. All of those features made the development process much more structured and easy to manage, especially when a need to refactor or even rewrite code arose.

With the pros also come the cons of such an approach. The codebase definitely grew larger due to some boilerplate code. It wasn't a huge concern since the Spring framework along with Lombok [6] also automated some processes, saving many additional lines of code. The execution and system response times might have also suffered a bit in comparison to if the service was written using the procedural programming paradigm. We were aware of this fact since the beginning and since it wasn't included in the project's requirements, we didn't stress about it too much.

Hexagonal Architecture

We knew right from the start that the core part of the system would be heavily dependable on other services and yet unspecified devices. It was difficult to start writing code without the proper knowledge on how those different instances would communicate with each other. That's when we decided to look for an architectural pattern that would help us start development with the business logic first in mind and worry about all the previously mentioned concerns later with the foundations already in place.

What we found is something that we had in mind while trying to figure out the solution to our problem, but described in detail and structure. The **Hexagonal Architecture** delivers on all of

our needs. It is a software design pattern first published in 2005 [5], that has been on the rise lately, that could potentially replace typical multitier architecture choices (such as a three-tier architecture).

The pattern divides the system into the following parts [17]:

- Domain - the most inner layer that describes the business logic
- Application - a layer that orchestrates the use of the entities from the domain layer
- Ports - they act as interfaces through which the application can interact with the outside world and vice-versa.
- Adapters - in the hexagonal architecture schema they belong outside of the core hexagon. They are technology-specific implementations that allow for interaction between the application layer and external services and devices. We can distinguish two types of adapters based on the direction of communication. Inbound adapters handle the incoming traffic - in our case the HTTP requests from applications. Outbound adapters handle outgoing traffic, which includes communication with devices as well as with the database and other micro-services.

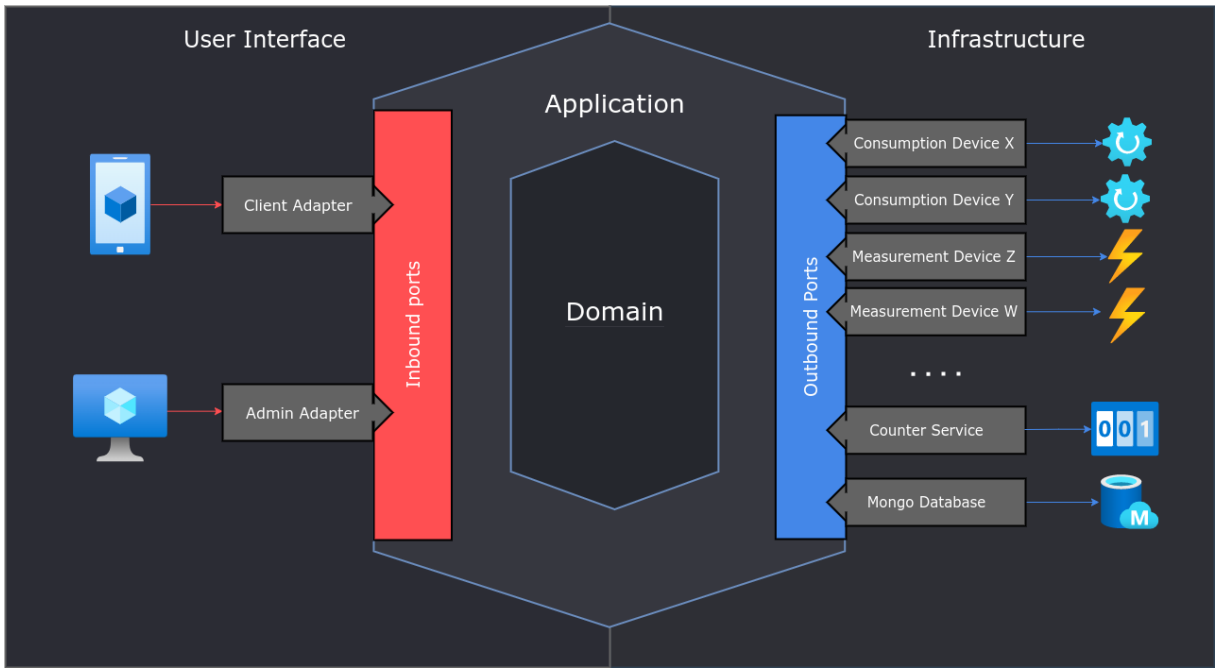

Figure 3: Hexagonal architecture schema

Different layers in our system communicate through the use of interfaces according to the rule of **dependency inversion** [2] with a special emphasis on not making the inner layers dependent on outer ones - only the other way around.

The Hexagonal architecture has traits of a Clean Architecture - a concept introduced by Robert C. Martin in his works. [20] The architecture of our Management service checks all of the most important boxes in regards to Clean Architecture [16]:

- Testability - our system works on real, physical devices therefore we needed a way to easily test it to make sure it behaves as we expect it. Decoupling the business rules from external concerns and dependencies allowed us to test those business requirements independently. Every layer is separated from the rest thanks to the use of interfaces, so the actual implementations of other components can be easily mocked. In consequence the unit tests are much simpler and faster.
- Technology agnostic - layer separation and ports and adapters usage enable us to choose different technologies to implement parts of our business logic. In our case it is particularly important, since the communication with different devices varies. It allows us to handle all of them without the need of changing any of the other layers. On top of that if a need for a new database ever arises, the change will only require a new adapter.
- Domain-Driven-Design [3] - since the domain is separated from the implementation details, we could immediately start focusing on the actual workings of our system and defer the technology-specific decisions to a more suitable time.
- Maintainability - changes to one module do not affect the others. It allows us to focus on one part of the project separately, which definitely speeds up the development process.

The use of the hexagonal architecture pattern was not easy, mainly because of its steep learning curve and the required knowledge of various possible other patterns that could work with our system. Nonetheless it allowed us to create a system that is easy to maintain and can be easily extended to handle new devices, which was our primary focus.

Technology

The code of the service was written in **Java** with the use of the **Spring** framework and managed with the **Maven** build tool mainly due to our previous experience in this technology.

The service was dockerized to allow for ease of use irrespective of the run environment. Local development services like databases were also containerized and run using Docker Compose.

The main libraries used in the service were:

- Lombok - a library that simplifies and speeds up the implementation process by replacing the need for writing boilerplate code with annotations.
- OpenAPI - with the use of the Swagger tool it provides an API documentation that proved to be very helpful during the integration with the frontend applications.
- Junit, Mockito - libraries used for testing
- Checkstyle - used for testing code quality in the CI environment

Usage

The hexagonal architecture greatly simplified the process of adding new types of devices. The basic requirement for their handling is a presence of a response structure expected from the device and a mechanism to map it to a domain object. Depending on the class of the device they are required to send information such as their status and a power on/off answer (consumer device) or current measurements (measuring device).

The code necessary to communicate with those devices must be included in the form of their own adapters implementing a system defined interface. To simplify the implementation process we have provided exemplary ones for the actual devices we used for the testing of our system. Both types of devices were Supla branded appliances: measuring - MEW01, consumer - ROW01

```
public enum ResponseType {
  1 usage
  SUPLA_METER_MEW01,

  1 usage
  SUPLA_CONSUMPTION_ONOFF_ROW01,

  1 usage
  SUPLA_CONSUMPTION_READ_ROW01;


  1 usage
  public static Class<?> stringToResponseClass(ResponseType responseType) {
    return switch (responseType) {
      case SUPLA_METER_MEW01 -> SuplaMeterMew01Response.class;
      case SUPLA_CONSUMPTION_ONOFF_ROW01 -> SuplaConsumptionRow01OnOffResponse.class;
      case SUPLA_CONSUMPTION_READ_ROW01 -> SuplaConsumptionRow01ReadResponse.class;
      default -> throw new IllegalStateException("Unexpected value: " + responseType);
    };
  }
}
```

Figure 4: Enum that connects device with parser

A response type must be known to the system for it to know how to handle it. When it's specified the system returns its actual class representation.

```
public record SuplaMeterMew01Response(boolean connected, Float support, Currency currency,
                                      Float pricePerUnit, Float totalCost,
                                      1 usage
                                      List<PhaseResponse> phases) implements MeasurementRead {

  @Override
  public MeasurementReadAction toDomain(MeasurementDevice measurementDevice) {
    Float measuredValue = (float) -phases.stream().mapToDouble(PhaseResponse::powerActive).sum();
    return new MeasurementReadAction(measuredValue, measurementDevice);
  }

  2 usages
  public record PhaseResponse(Integer number, Integer frequency, Float voltage, Float current,
                              1 usage
                              Float powerActive, Float powerReactive, Float powerApparent,
                              Float powerFactor, Float phaseAngle, Float totalForwardActiveEnergy,
                              Float totalReverseActiveEnergy, Float totalForwardReactiveEnergy,
                              Float totalReverseReactiveEnergy) {

  }
}
```

Figure 5: Measurement Data Transfer Object

The use of Java 18 allowed us to take full advantage of a fairly new feature in the form of records, which made the implementation of POJOs (Plain Old Java Objects) quick and simple.

```
public record SuplaConsumptionRow01ReadResponse(Boolean connected, Boolean on,
                                                Boolean currentOverload) implements ConsumptionRead {

  @Override
  public ConsumptionReadAction toDomain(ConsumptionDevice consumptionDevice) {
    return new ConsumptionReadAction( isOn: connected && on, Action.TURN_ON, consumptionDevice);
  }
}
```

Figure 6: Consumption Status Data Transfer Object

```
public record SuplaConsumptionRow01OnOffResponse(Boolean success) implements ConsumptionOnOff {


  @Override
  public ConsumptionOnOffAction toDomain(ConsumptionDevice consumptionDevice) {
    return new ConsumptionOnOffAction(success, consumptionDevice);
  }
}
```

Figure 7: Consumption Switch Data Transfer Object

The handling of newly introduced functionalities is also possible. However such features require some changes in the domain part of the system, through implementing the business logic and making changes in the device's adapter to handle the new responses.

```
public interface DeviceGateway {

  5 usages  1 implementation
  <T> T request(Device device, Action action);

  1 usage  1 implementation
  ConsumptionReadAction requestDevicesStatus(ConsumptionDevice consumptionDevice);

  1 usage  1 implementation
  MeasurementReadAction requestMeasurement(MeasurementDevice measurementDevice);

  1 usage  1 implementation
  MeasurementReadAction requestOnOff(ConsumptionDevice consumptionDevice, Boolean turnOn);
}
```

Figure 8: Device Gateway - port representing device management

Functionalities that don't require any additional logic also do not force a change in the code. It's sufficient to simply add the details of their connection to the database and the management algorithm will call upon them when it decides they should be performed. The different functionalities can be seen on the Fig. 9 in the "endpoints" part of the JSON.

Figure 9: Representation of Consumption Device Document in MongoDB Compass

3.2.4. Authentication service

To authenticate traffic going through our product, we have decided to use **Firebase** in the whole system. It guarantees safe access to resources between applications, manages user accounts and their sign-ins and sign-outs.

The service is used by all components of the system - the management service and the web and mobile applications. The libraries used in all of those systems support the use of the Firebase service, which made the communication much easier.

3.2.5. Computing service

Since we didn't want to put a huge strain on our request limits and the devices themselves we had to space out our API calls and create a service that would fill in the missing data. The Computing service has a couple of use cases, one of it being calculating the integral from the provided data.

Since the service is small and its functionality really specific, we have decided to use **Python** to implement it. Python is extremely useful for such tasks - it enabled us to scrape up a functioning service in a short time, which ultimately made it to the final version of our product.

The use of Python was also backed up by one of its most used libraries in **scipy**. From the variety of methods of integration provided by the library we have decided on the *Simpson's rule*, which is particularly good for calculating integrals of curves [13]

3.2.6. Simulation tool

It is a mix of the the **management service's** functionality combined with a **Python** script for chart plotting. The idea of the component is to simulate the workings of the management service. Running the service on a production system without a prior test might cause some unwanted behaviors. That's why the tool can use historic data from the production database, copy it to a test environment and run the management service, fully simulating the backend part of the system.

After the test we can use a script to plot the returned results after the system finishes its run and analyze them to make sure that no mistakes occurred during the process.

3.2.7. Database

As our database we have decided to use **MongoDB** - one of the most popular choices on the market. We decided to go with a non-relational database, because the database itself isn't complicated. We do not perform any advanced operations including multiple tables, so the need for a typical SQL database never arose.

User friendliness and a wide range of support tools such as **MongoDB Compass, MongoDB Atlas** and **MongoDB Charts** were also a good argument for choosing MongoDB as our go-to database.

MongoDB Atlas is a Database-as-a-Service utility, which takes away the responsibility of managing the database by the user and is instead stored in the cloud. It guarantees all the benefits of cloud computing, including scaling and high availability. By default it also creates replica-sets in order to help in the unfortunate event of a need for disaster recovery.

MongoDB Compass is a desktop application used for database querying and data management. MongoDB Charts is a tool used for data visualisation.

During the development process we often needed to create small, reusable databases for different purposes. Thanks to all of the above mentioned tools, the process was much easier. If changes were being made to the database itself we also needed to perform database migrations - some of the data acquired during the development process was already very important for the final results of our works. To migrate the data we used script languages such as **Python, JavaScript** and **bash**

3.3. Deployment

During development we used **Docker Compose** to easily setup most of the necessary services and applications.

In the production environment however both the Management service and the web application are run on **Heroku**, which provides a **Software as a Service** solution. [21] The software is deployed on the cloud and run from there. By doing so we take away the responsibility from the user of providing sufficient infrastructure to meet the system's demands and don't force him to constantly check on the status of the set-up.

The database is being run in a similar fashion using the previously mentioned MongoDB Atlas service.

The mobile application is the only piece of software that we expect to physically be on a device. It is exported to an apk format so it is available for download and installation on smartphones.

3.4. Summary

In this chapter we described our approach to the technological aspects of the system, the design decisions that we made and problems that we encountered along the way.

4. Work organization

In this chapter we will dive into the process of creating our product, how we planned out our work and what each person in the project was responsible for. We will also cover used programming techniques and our approach to designing and testing the system.

4.1. Project's characteristic and implementation

The idea for the project came from one of the authors of the system. We were contacted by *Góralmet Sp. z o.o.*, which required a simple system that would allow for autonomous energy distribution. After gathering enough information about the expected product we decided to pair up and visited Doctor Sławomir Zieliński to propose the idea as the topic of our graduation project. After discussing some of the more important aspects of the product we came to an agreement and started thinking about the initial system's design.

4.1.1. Product expectations

To meet the expectations of both parties (Góralmet and Doctor S. Zieliński) we had to carefully plan out our entire workflow and prioritize different aspects of the product at different stages of development.

Góralmet required a simple solution, providing only the bare minimum necessary for the system to work. The main focus was on the already present devices at their facilities that were expected to perform only one functionality. Time was a big factor here as the most beneficial time for photovoltaic energy production is during the summer, so we wanted to have this version of the product ready as soon as possible. The versatility of the system when it comes to different devices wasn't required, neither the mobile or web application. As developers we had an easy access to the on-site database which also made it unnecessary to implement any additional features.

Doctor Zieliński on the other hand expected a fully fledged out product with a frontend interface that a non-technical person could operate and manage the system through. The most important feature of the system was supposed to be its versatility in managing different types of devices with various ranges of functionality. The web application was supposed to be used as a tool to manage the product through and the mobile application as a means to watch over the system. Because of all of the additional requirements, we knew right from the start that the time necessary to implement them all would be a lot longer compared to the simple version.

4.1.2. Project phases

The way in which we developed our project wasn't set in stone and changed as needed along the way. During the beginnings of our work we adopted a model of prototyping solutions and deciding on what we liked and didn't like, while continuously discussing our decisions with both client parties. When the basis for our project was already laid out, we then moved to a more organized and strict way of thinking about the further development tasks. We started working in a **waterfall model** [10], since most of the requirements for the project were specified in detail and only left to us to implement.

The project can be separated into the following phases:

- Planning - during this phase we created the vision for our product and then confronted it with the expectations of both client parties. It was here where most of the requirements for the system were defined.
- System design - during this phase we decided on the technology that we would use and made decisions regarding the architecture, inner workings and features of the system.
- Implementation - the longest and most time consuming phase of the project. That's when most of the code was written and ideas were implemented.
- Testing - a phase during which we tested the system and its features.
- Deployment - the last stage of development, during which we could officially deploy our product.

Despite our best attempts to stick to the vision created in the initial stages of development the specification of the project made it nearly impossible to do so. Because of difficulties or small changes in requirements some features were at times reworked or redesigned. What made us stray away the furthest from a strictly waterfall model was a not so rare need for creating and implementing new tasks, which we didn't predict before. That's when our work style used to temporarily change into a more agile based methodology. In the implementation phase we used an incremental process, which means we focused on specific modules or functionalities and worked on them until they were done.

The implementation and testing phases were quite mixed up, as most of the testing was already done during the implementation stage. While writing code and testing its functionality, most bugs were caught and immediately fixed. Even towards the end of the implementation stage we started to routinely check the system's working to make sure none of the previously not-connected components were not interfering with each other. Therefore the testing stage was mainly used for overall testing of the already finished product and implementing minor fixes.

4.2. People in the project

We worked on the project as a two-man team and decided to strictly divide our scope of responsibilities right from the beginning.

- Jakub Janicki - Full-Stack developer, DevOps, mainly responsible for the server side of the system and the corresponding services. Also involved in the web application.
- Adam Przywieczerski - Frontend Developer and Product Owner, responsible for implementing the mobile application as a whole and in part the web application.

4.3. Responsibilities within the team

We tried to divide the work among us in a way that would best suit our abilities and allow us to develop different parts of the system independently. Hence the almost complete division in the backend part of the system, where Jakub implemented most of the functionalities and Adam only reviewed them. The roles in the creation process of the mobile application were reversed. The work on the web application was divided among the two of us, as it was the last part of the system to be implemented. By that time we were mostly finished with all of our previous tasks.

The most important tasks done by us are:

Jakub Janicki

- Server
  - Management module
  - Measurement module
  - Consumption module
  - API supported by the server
  - Simulation tool
  - Micro-service for computational purposes
- Web application
  - Sign in/up view
  - Main dashboard view
  - About us view
  - Linking the application with the server
- Miscellaneous
  - Heroku server setup
  - CI/CD configuration for the server
  - Mongo database setup
  - Firebase setup

Adam Przywieczerski

- Mobile application
  - Login screen
  - Home screen
  - List of active devices
  - Chart of summed up energy surpluses from all farms
  - Side menu and its content
  - Measuring devices screen with miniature views of devices
  - Individual measuring device screen
  - Measured energy chart
  - Measuring device information
  - Consumer devices screen with miniature views of devices

  - Individual consumer device screen
  - Consumer device activity chart
  - Consumer device information
  - Linking the application with the server
- Web application
  - Measurement devices - details and management view
  - Consumer devices - details and management view
  - Users - details and management view
  - Farm control - details and management view

4.4. Work organization and tools used

During the entire project's workflow we mainly communicated with the use of the **Messenger** application. It allowed for quick communication in topics that didn't require to much background or previous knowledge of certain issues. In cases where the problem was much larger or harder to explain we often resorted to simple phone calls or voice chats such as **Discord**. Each time when we needed to make a big decision regarding the entire project we either met in person or discussed it through voice communicators. Despite not having a set time for team meetings, we gave each other updates on a daily basis.

The meetings with Doctor Zieliński varied in frequency. During the first semester of our works we met every 2 weeks on average and during the second semester once a month, in both cases through the use of the **Webex** platform. Each meeting and the topics discussed during them were recorded in the form of *minutes*. The meetings with our client from Góralmet mainly took place at the company's headquarters, since they required not only customer's and developer's presence, but also involved their engineers. Minor meetings were performed remotely. We also regularly provided our client with statistics data by sending emails using **Gmail**.

As our source code management tool we decided to use **Git** and **GitHub** as the service on which we hosted our code. We both use those tools on a daily basis, so it was the most natural choice for us to make. In tandem with each other they provide a range of capabilities that make the development process much easier. The use of feature branches allowed us to keep the main branch clean at all times and only update it when new functionalities were ready to be merged. GitHub Actions made the merge possible only when all CI checks were met. GitHub through the use of **Github Projects** [4] acted as the main tool in organizing our workflow. As there were only two of us we decided against using any of the bigger tools like **Jira** or **Trello** to not unnecessarily complicate the whole process. GitHub projects provide a really neat and easily manageable environment for controlling the workflow. Each user story created in the project could be linked to any of our repositories as an issue along with the pull request in which it was implemented. To better organize our work we divided the tasks into 4 categories:

- Backlog - defined tasks that were needed for the completion of the project. Each time a new feature was required it was put in the backlog.
- ToDo - tasks chosen for implementation in a specific iteration
- In Progress - tasks currently being worked on

- Done - tasks that were successfully merged into the main branch

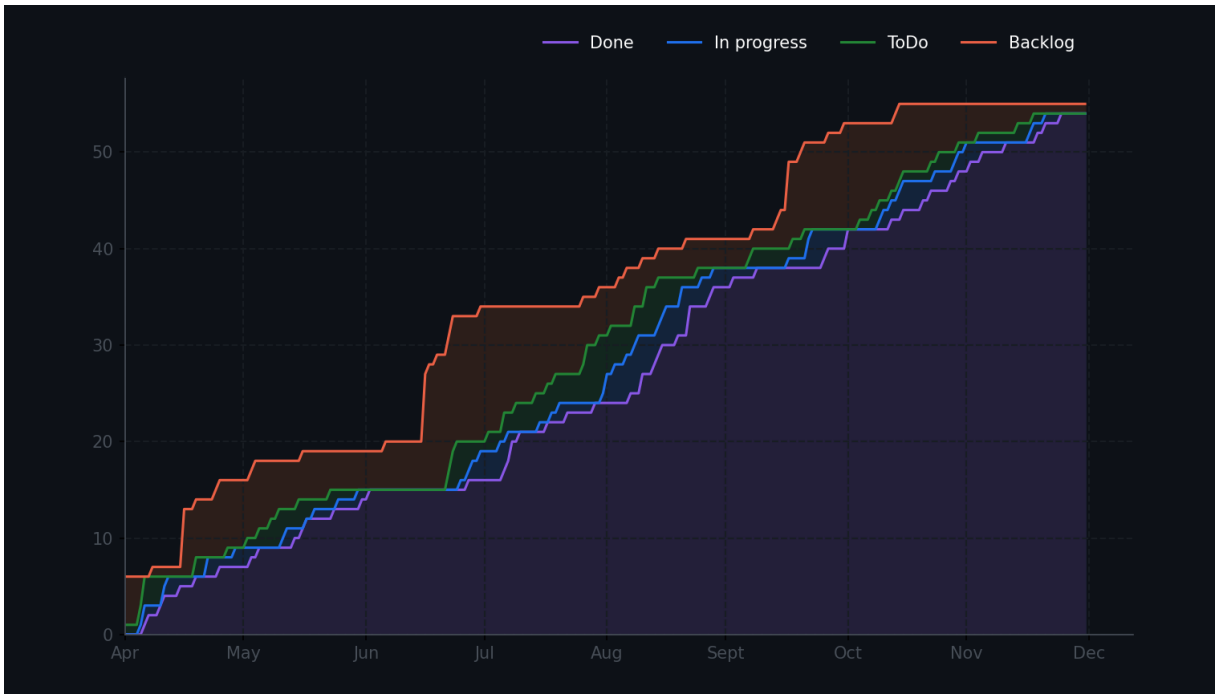

Figure 10: Task statuses on GitHub projects

On the Fig. 10 we can see the progress of our works divided into singular tasks. We started off in April with a lot of planning work, so the tasks were rather short and concise. Through May and early June we kept working on the initial version of the system. What is rather noticeable is a pretty uneventful late June and July. The lack of activity was mainly caused by final exams and the beginning of summer holidays, during which we wanted to rest a bit before continuing the works. In August we again started working hard on the system and tried to keep the pace until the end of the year. Along the whole timeline we can also see a few different points, where our backlog visibly grew in a short period of time. Those were key moments during which we decided on our next steps in the project and planned out at least a part of the upcoming work.

To make the process of backend/frontend integration easier we decided to use the **OpenAPI** library. As a result our server auto-describes every API route that is handled by the system's controllers. Every endpoint description consists of a route, parameters, a request body and an exemplary request. We additionally used **Swagger**, which provides a user interface for all of the endpoints. The Swagger web application is being hosted by our server whenever it is running.

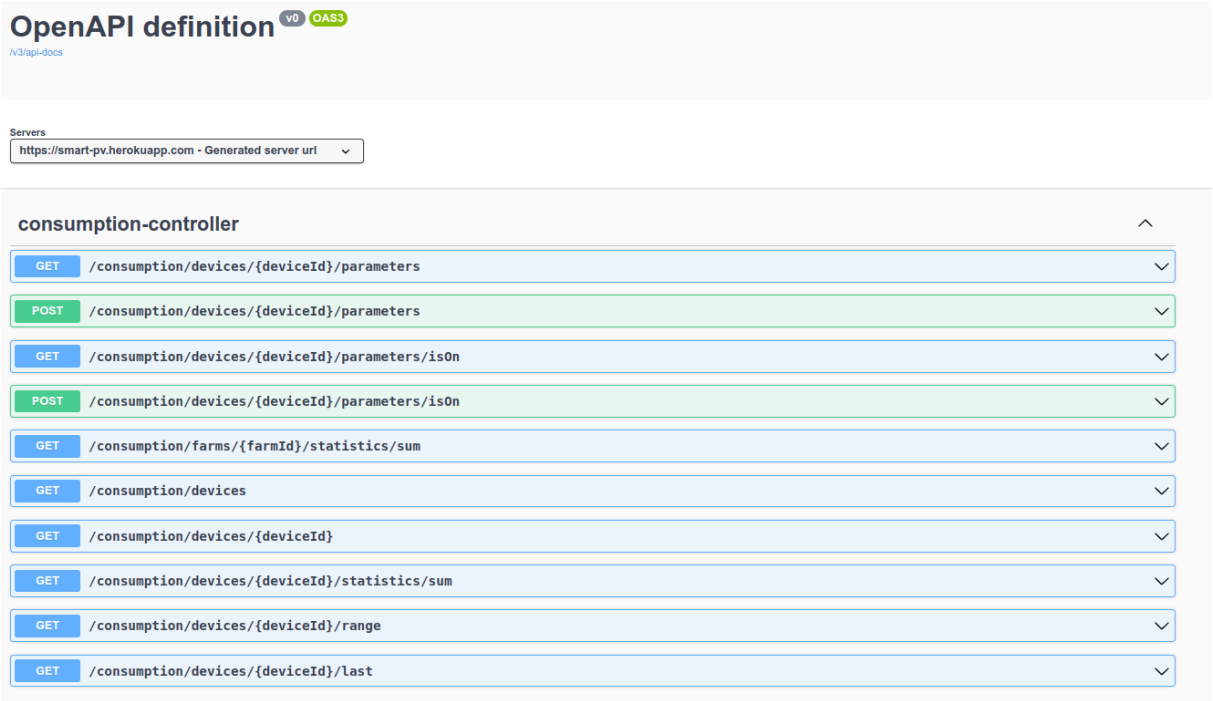

Figure 11: Part of endpoints' documentation in Swagger

4.5. Used techniques and practices

During the works on the project we tried to apply some of the industry practises into our development process. One of the most notable ones was definitely **Pair Programming** [14]. As there were only two of us, we fairly frequently sat down and worked on a problem either of us was facing. Trying to work through it, we often caught the mistake that the other person was making or sparked new ideas in places where it was difficult to make progress. It was definitely the most rewarding practise, as it allowed us to peek into the details of each other's work and made us work closer as a team.

The other practise that we performed was **code review**. It allowed for catching up with the latest changes and made them safer to introduce to the main branch after both of us had looked at them. It was inseparably tied in the way we interacted with our repository hosted on GitHub. The development process was structured in the following manner:

- Feature branch creation - each new feature had its own separate branch.
- Implementation - the process of writing code by a person responsible for the task.
- Pull request creation - having finished working on a certain feature we pushed our code to GitHub and opened a pull request.
- Code review - the person that didn't write the code would check the changes introduced by the pull request and after a revision accepted them or requested changes.
- Merge - having accepted the pull request, with all CI checks passing, we merged the code to the main branch. After the merge there was also an automatic deployment performed to a staging environment.

Despite our best efforts to consistently do code reviews of all pull requests, we didn't require them for the merging process. Since there wasn't anyone else who could review the changes but

the two of us, we didn't want to force the other person to wait unnecessarily long to introduce simple changes in case any of us were unavailable.

4.6. Course of work

To better structure the course of our work we divided it into 4 separate stages. Each of them had a different goal in mind and helped us prepare accordingly.

4.6.1. Stage 1 (April, 2022)

It was the initial stage of our works. Having chosen the topic of our project, we started expanding our vision of the system along with the needs of our clients, while also narrowing down specific requirements. During this stage we did not yet start the implementation process. Most of the time was spent on meetings among ourselves and our clients.

Achieved goals

- Learnt customer needs
- Mapped out the work plan
- Planned out the development environment
- Technology selection
- Libraries selection
- Design of the farm management algorithm
- Design of the system architecture based on gathered requirements
- Designed database schema
- Initial design of the mobile application's UI
- Initial design of the web application's UI
- Suggested possible problem solutions to Góralmet and reached an agreement

4.6.2. Stage 2 (May - June, 2022)

This stage can be best described as the "prototype stage". It was here when we started implementing the first functionalities of our system and figuring out the complexity of the tasks before us. At the end of this stage we had the base of our server and a simple version of the mobile application.

Achieved goals

- CI/CD setup
- Server - creation of the programming environment - local, staging, production
- Server - attaching necessary libraries
- Server - creating hexagonal architecture

- Server - implementing configurations
- Server - implementing database schema
- Mobile Application - basic view of the application
- Mobile Application - simple communication between the app and the server

4.6.3. Stage 3 (July - September, 2022)

During this stage things started picking up pace. We deployed some of our architecture on cloud servers and had them running constantly. The previously prototyped server was turning into an actual backbone of our system and the mobile application started filling up with custom components and views.

Achieved goals

- Server - implementing the domain logic along with the management algorithm
- Server - implementing data collection from devices
- Server - implementation of the simulation tool
- Server - testing
- Server - communication with IoT devices
- Server - deployment of the server to Heroku
- Server - gathering statistics from the working system
- Mobile Application - different ways of navigation
- Mobile Application - view of devices
- Mobile Application - overall styling of the application
- Mobile Application - communication with the server

4.6.4. Stage 4 (October - December, 2022)

The last stage of our works was probably the busiest. Not only did we have to finish the server and the mobile application, but also start working on the web application. Alongside the implementation process we also had to regularly complete consequent chapters of documentation. The main goal for us was to successfully finish all of the tasks that we set out to do.

Achieved goals

- Implementing authorization on all services
- Server - finishing works on the API
- Server - implementation of admin operations on the system
- Mobile Application - data fetching from the system
- Mobile Application - data display on reactive charts

- Mobile Application - optimization of time consuming functionalities
- Web Application - base application
- Web Application - implementation of the measuring devices management view
- Web Application - implementation of the consumer devices management view
- Web Application - implementation of farm management views
- Web Application - implementation of user management views
- Web Application - optimization, testing, deployment to Heroku

4.7. Summary

The process of creating this project wasn't easy, but along with good planning and use of correct techniques and practises we managed to finish it in time. What helped tremendously and pushed us forward was the constant communication between the two of us and the general enthusiasm for our work.

5. Project's results

In this chapter we will present the results of our work. We will also summarize and give our thoughts on the whole development process. On top of that we will show the system from the perspective of its user interface, but also showcase the data generated through the use of the product. In the end we will also present our view on the future of the project, how it can be further developed or what things we wish were done differently.

5.1. Functionality overview of the final product

We accomplished all of the planned functionalities presented in the **2.2.** subsection except for the one presented in the **would** category. As stated from the beginning it wasn't imperative for us to include this functionality although it definitely would have made for a nice enhancement to the overall user experience.

The final functionalities of the system reserved only for the admin are:

- creation of the admin account and its management
- management of normal users accounts - creation and deletion
- farm creation along with the later addition of devices
- CRUD operations on the devices in the system
- task defining for consumer devices
- choosing between different energy consumption algorithms - one based on ranges of available energy, the other based on device priority

The final functionalities for the normal user (which also includes the admin account) are:

- logging in to the application(s) - normal user only has access to the mobile application, while the admin can also access the get application
- access to device's measurements gathered by the system and the activity history of the devices in the form of graphs
- overview of the farm's devices - status, currently performed task

5.2. Main usage scenarios

To show the full effects of our works we decided to present how one would setup his own system through the web application and monitor it with the use of the mobile application.

5.2.1. Sign up an admin account

In the login screen of the web application we have the option to create a new account by clicking the *Sign up* link as presented on the Fig 12. image.

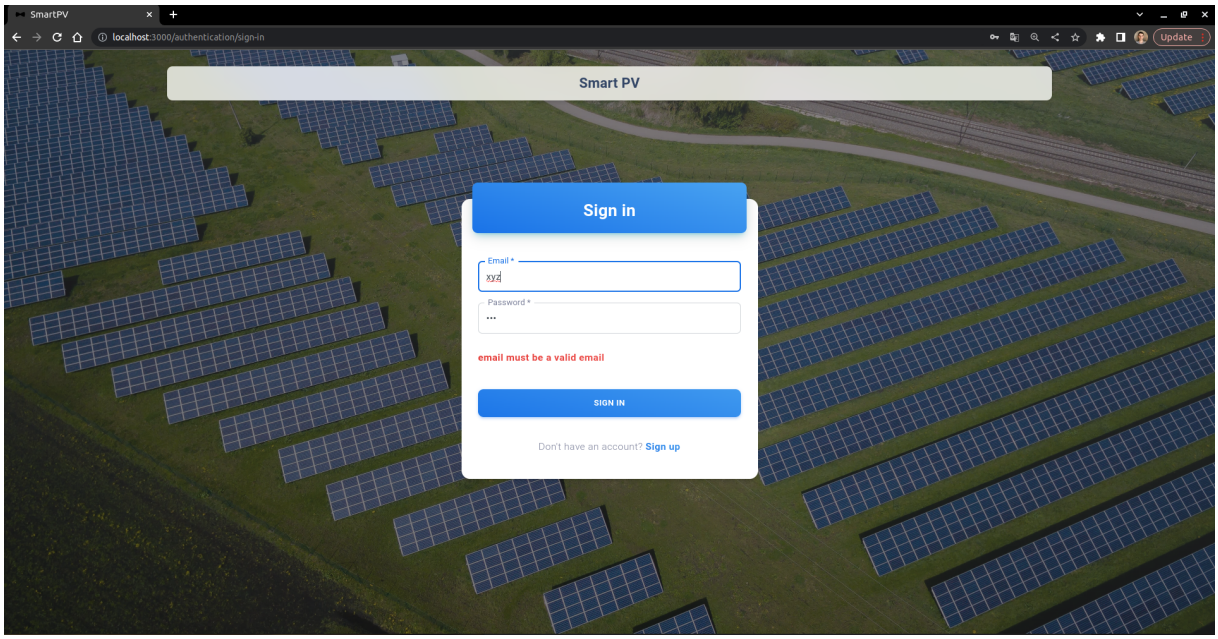

Figure 12: Sign in screen - web application

What happens next is a fairly standard way of signing in to a new service. We provide our username, an email address and a password to our new account.

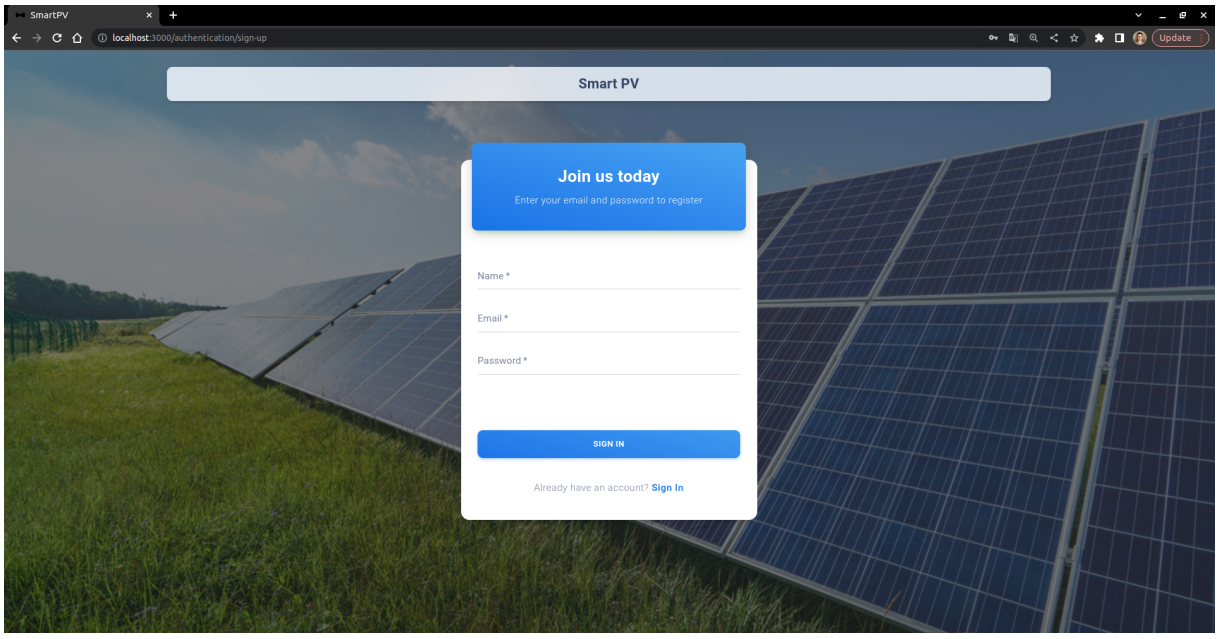

Figure 13: Sign up screen - web application

After the account has been created we can go back to the *sign in* screen and use our new credentials to log in to the web application.

What we accomplished by this process is we created the admin account which will be responsible for the management of an instance of our system. Through this account a farm owner has complete control over the delivered product.

5.2.2. Farm creation

Farm creation is the first thing any user of the system would want to do. It allows for connecting all of the later added devices into a single network and managing them through it. Each farm has its own set of devices, a currently ongoing energy distribution algorithm and data gathered over time. The creation screen can be seen on fig. 14.

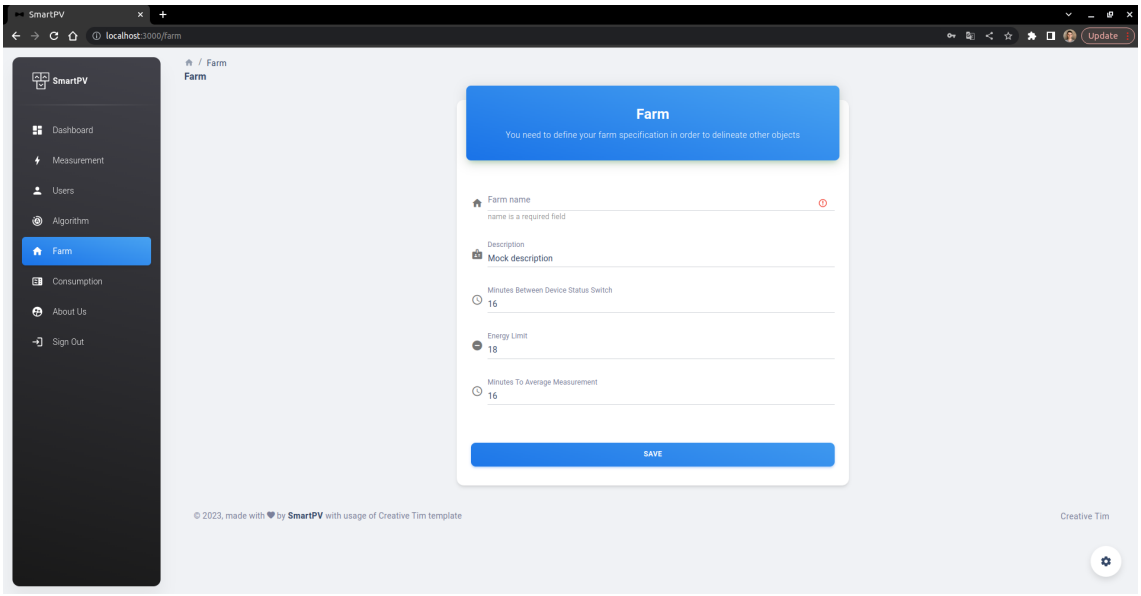

Figure 14: Farm creation - web application

5.2.3. Users management

If we want more users than just the admin to have access through the mobile app to the metrics generated by the system, we must create accounts for them. In a separate tab called **Users** we are presented with the list of currently existing accounts.

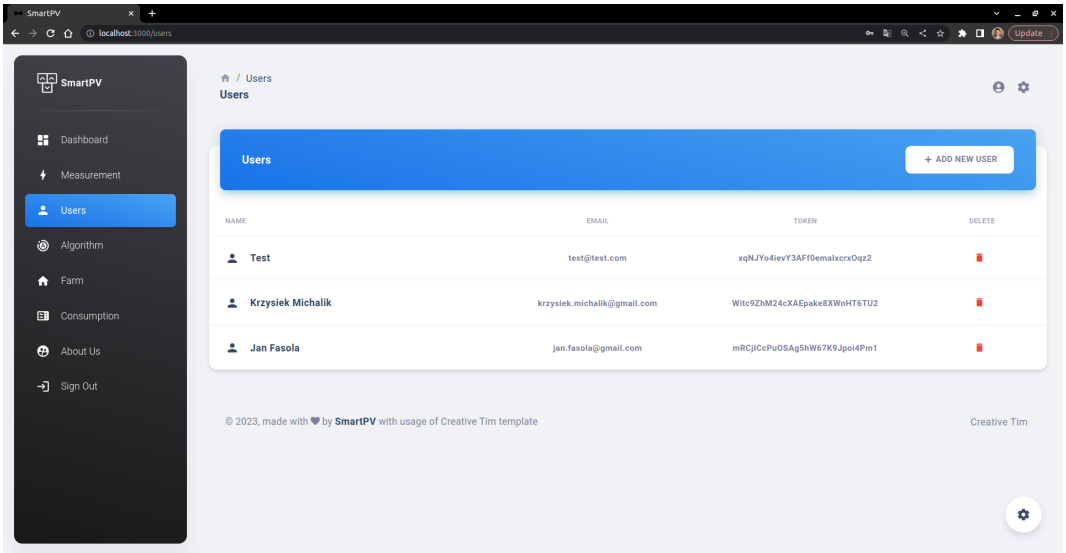

Figure 15: Users - web application

After clicking the **Add new user** button we are taken to the screen, where we are prompted to enter the new user's name, email address and password. After filling out all information we create the new account and can immediately see it in the list.

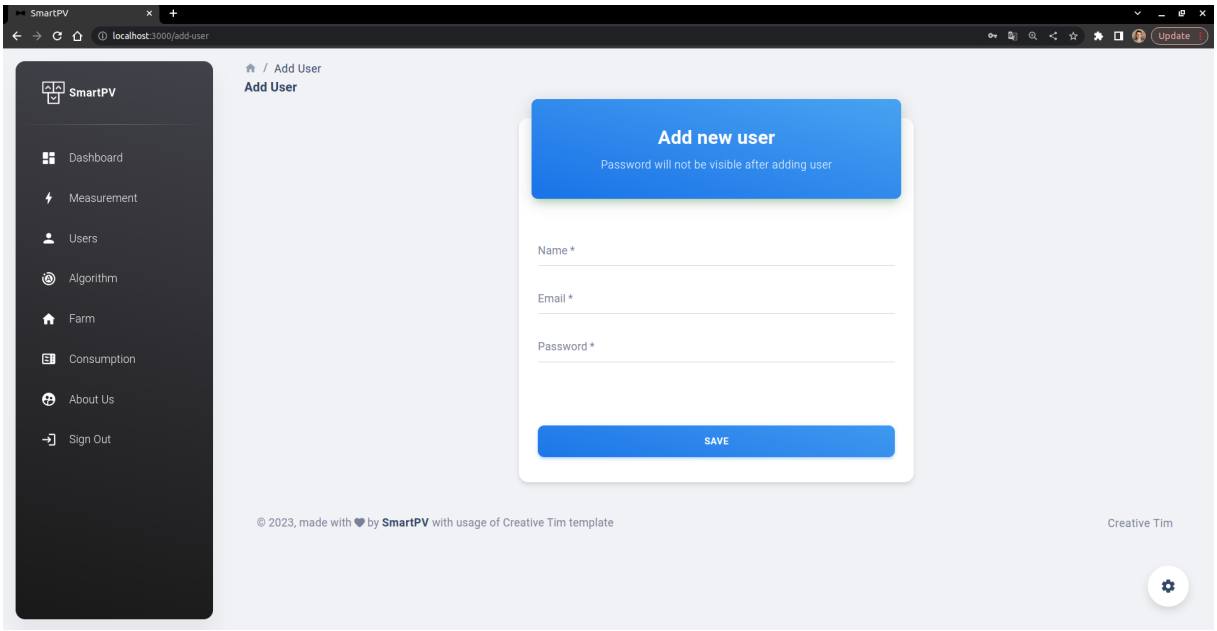

Figure 16: Add new user - web application

5.2.4. Devices management

Both screens showcasing the devices (consumer and measuring) are similar with the same mechanism of adding new devices to the system. In the Fig. 17 we can see the screen of measuring devices with the creation button in the upper right corner.

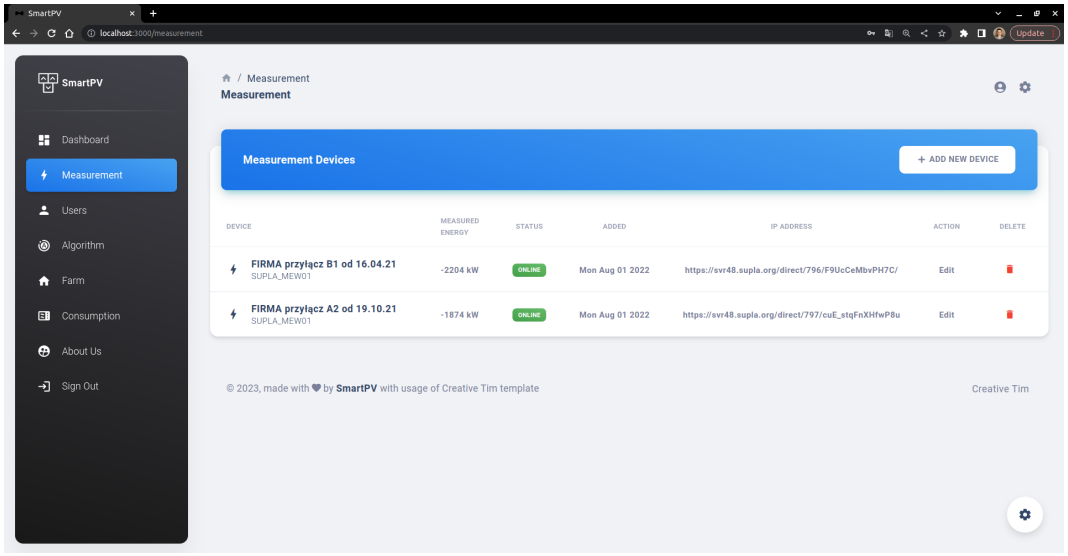

Figure 17: Measuring devices - web application

Next to each device there's also a trashcan icon that allows the admin to delete the device from the system. A simple click anywhere else in the row takes us to the creation/edit screen of a

particular device - they are the same screen, but in the case of editing a device all information is already filled out.

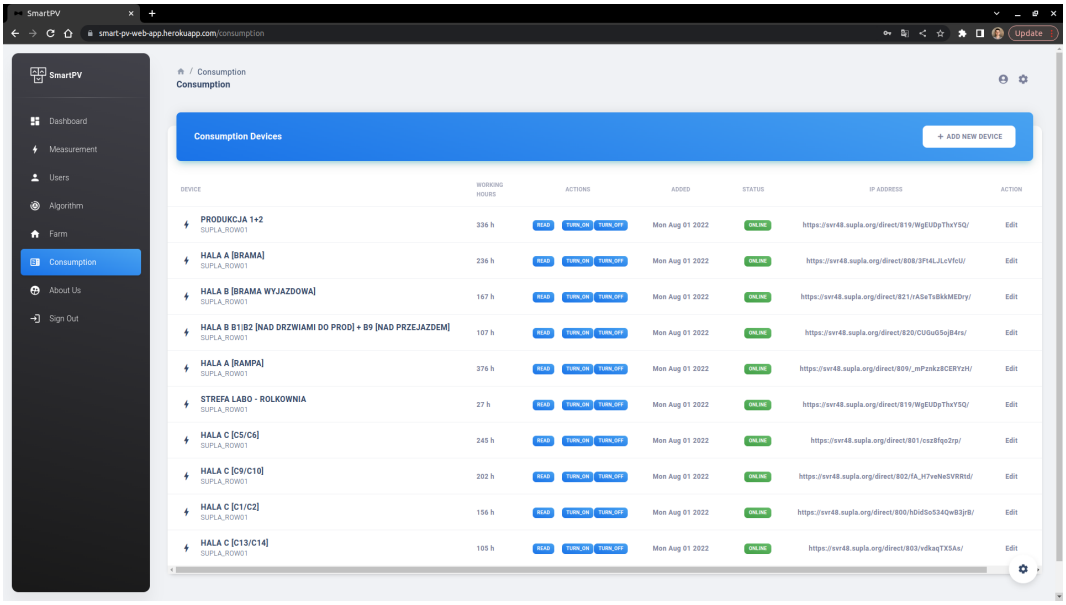

Figure 18: Consumption devices - web application

5.2.5. Addition of a measuring device

After deciding on adding a new measuring device to the system we are taken to the following screen.

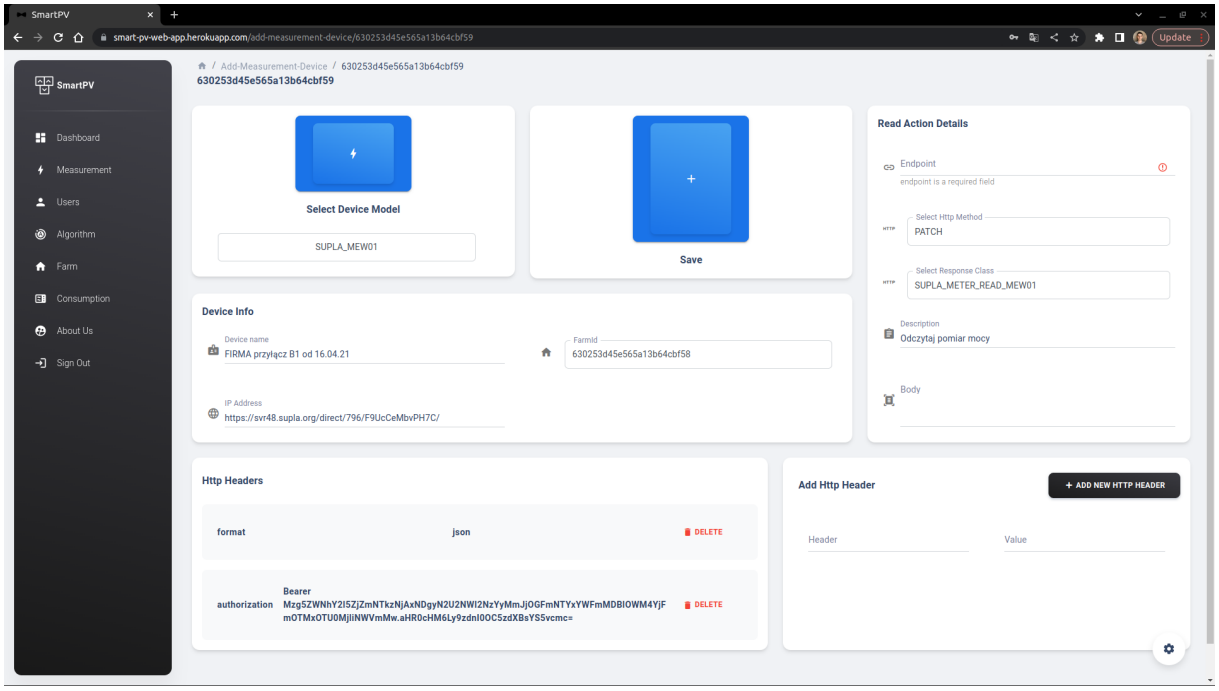

Figure 19: Add measuring device - web application

To add a device we must provide the necessary information, except for the endpoint description and additional HTTP headers, which are optional. Each device must have a name, a chosen model, a farm to which it is bound, IP address and an endpoint.

5.2.6. Addition of a consumer device

Similarly the process of adding a new consumer device to the system takes us to the screen seen on Fig. 20 and consists of a few steps.

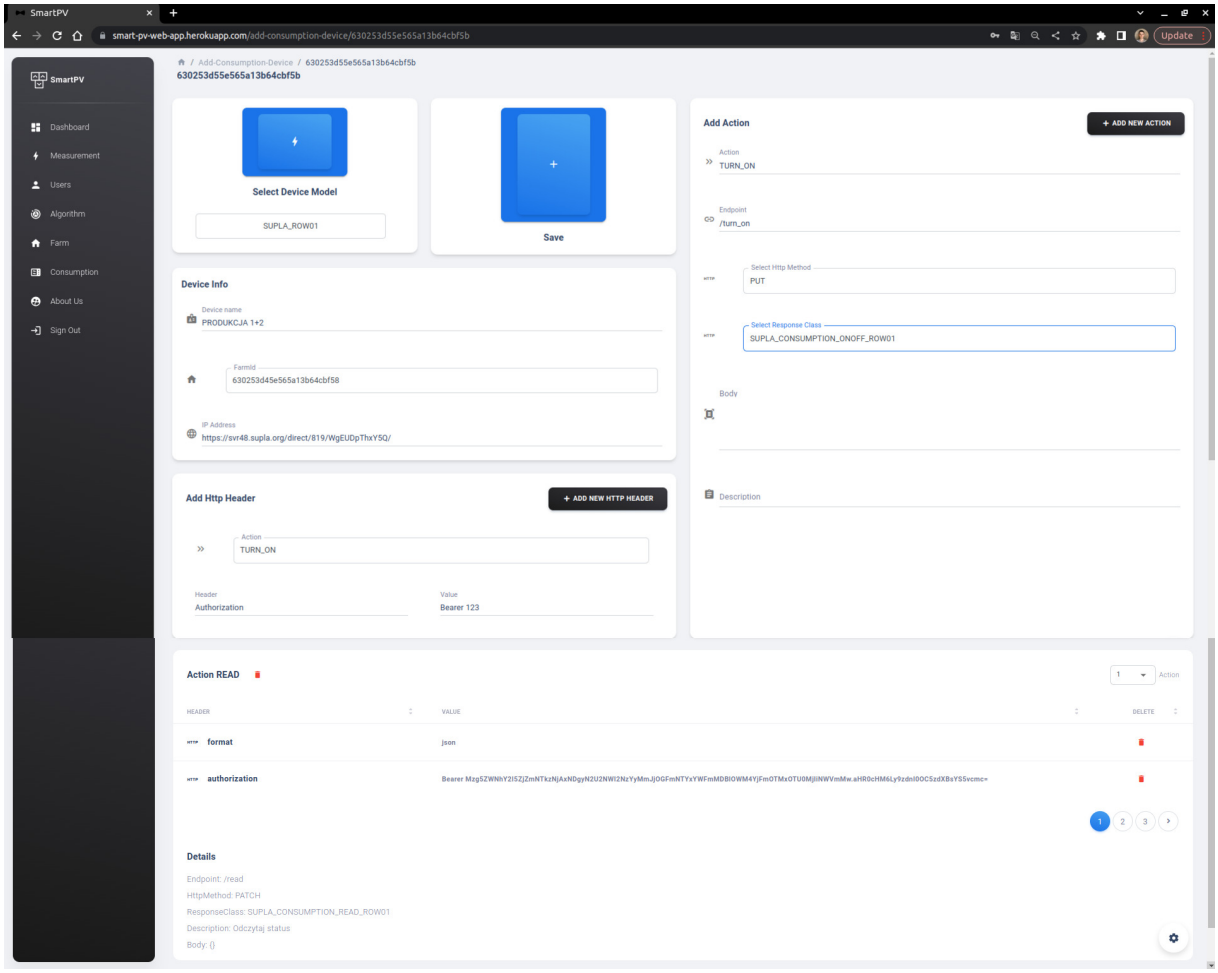

Figure 20: Measuring devices screen - web application

As was the case with the measuring devices we must choose the appliance's model, input its name, IP address and id of the farm it belongs to. Each device holds its own set of functionalities in a list. In the same screen we can define and add new functions to this list by providing its name (endpoint), HTTP method, header and body and a response class - a data transfer object for the server to understand the response from a particular type of device.

5.2.7. Choosing the energy distribution algorithm

In the **Algorithm** tab we can pick the algorithm that we want to run for a specific farm.

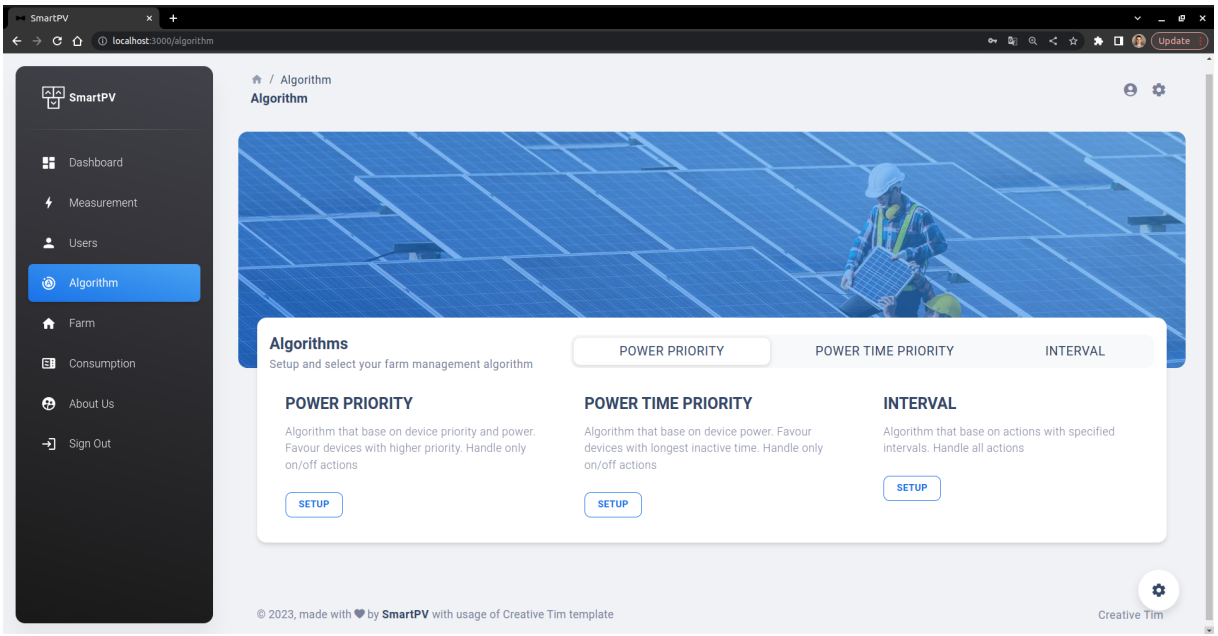

Figure 21: Algorithm selection - web application

After choosing we then proceed by clicking the **Setup** button under the selected algorithm type. This takes us to the configuration screen which is different for every type of algorithm. Each algorithm has a parameter, which regulates the waiting time between performing two actions on the same device. It minimizes the risks of overworking devices. The algorithms also all have a minimum energy threshold, below which they will not perform any actions. The user can also choose how often the algorithm will be run. It might often be aligned with how frequent the measurements in the system are done to maximize the usage of devices. We provided 3 algorithms in the system, each with its own premise.

Priority algorithm

It provides a simple configuration process, but restricts users to using only a basic "ON/OFF" command on each device. It is the simplest to setup and doesn't require a lot of thought put into it. If the customer knows the exact energy consumption of a device, is sure that it doesn't change over time and wants the devices to work in the same way all the time, then he will probably choose this algorithm. Each device is selected by taking into account its priority and energy usage. If there are at least two available devices with the same priority the one to be run is selected randomly.

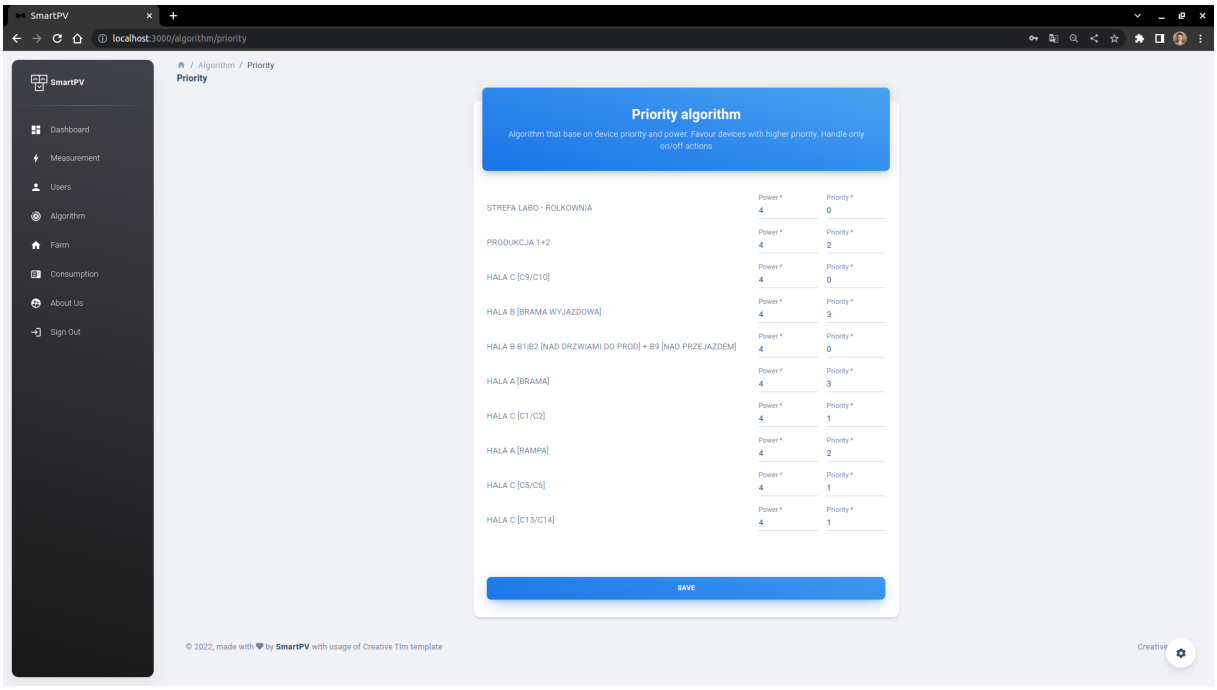

Figure 22: Priority algorithm configuration - web application

Interval algorithm

It is the most complex algorithm to set up, but it also provides the widest range of possibilities for the customer. It enables users to add different functionalities for devices on top of the default "ON/OFF" approach. The operation of the algorithm bases itself on the use of editable intervals. When the energy surplus reaches a certain level, each task defined in the corresponding interval is activated in the system. The lower and upper bounds provide a bit of leeway in the event of minor spikes or drops in energy readings. The user can add multiple intervals (it is impossible to add overlapping intervals) and then fill them with different tasks, each with its corresponding device and action.

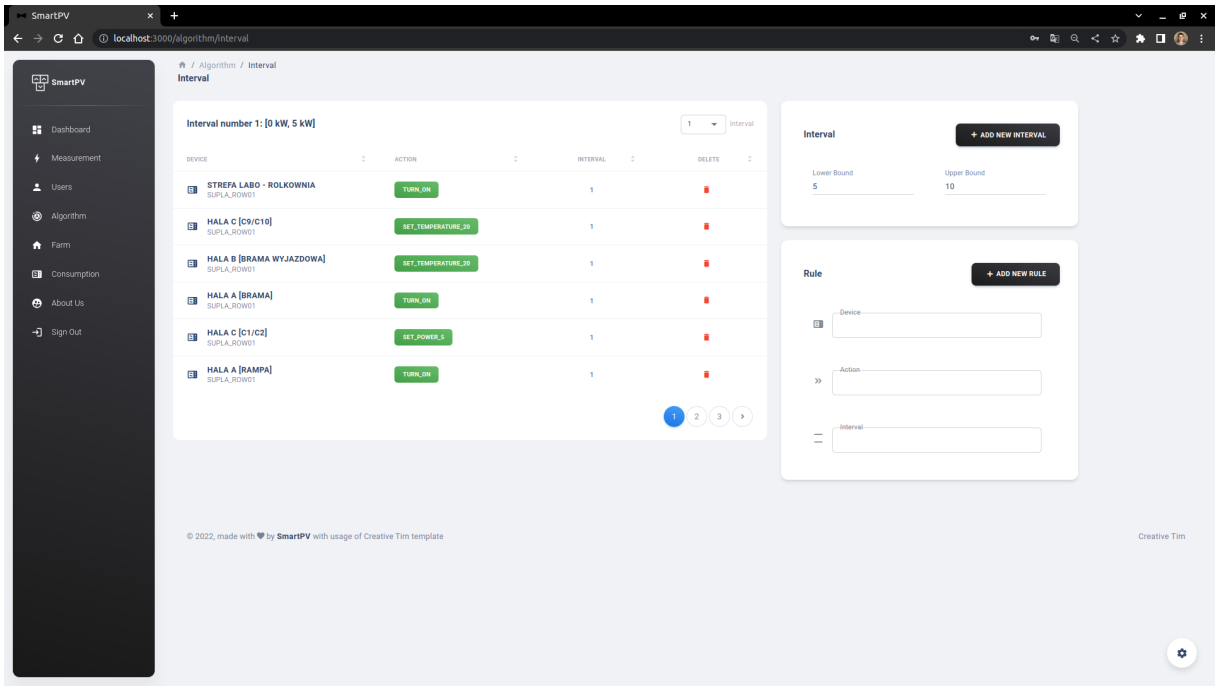

Figure 23: Interval algorithm configuration - web application

Time priority algorithm

It is very similar to the Priority algorithm, sharing only the ability to turn devices on and off, but has a different goal in mind. We do not prioritize a certain set of devices over another, but rather treat them all equally. We choose the appliances to run based on the last time they were used. The further back in the past, the more likely a device is to be selected. Due to this approach, we can perform a regular checkup on all of the devices and keep them in good condition. During the configuration, we only have to provide energy consumption levels for each of them.

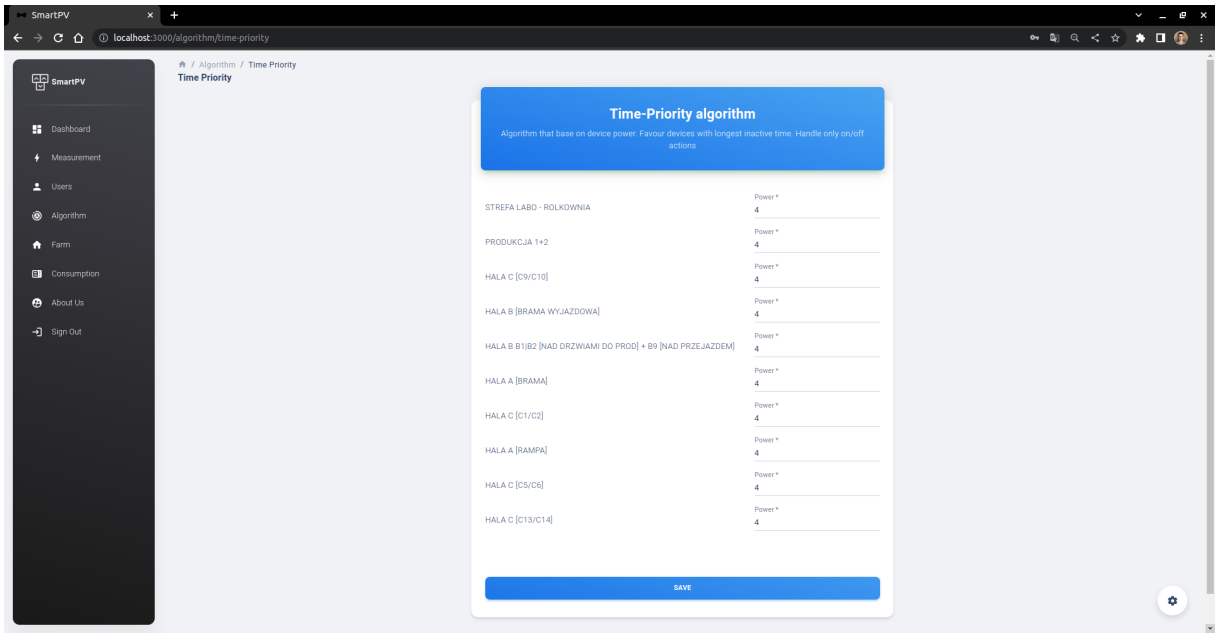

Figure 24: Time priority algorithm configuration - web application

5.2.8. Running the system

After filling out all of the necessary information in the previously mentioned tabs, the user can finally get to the main focus point of the system - running it with the underlying algorithm of choice.

In the **Dashboard** tab, the administrator can see all of the prerequisites for running the system. If the icons in all of the boxes have a green background, then the system is ready. In case one of the boxes is lit up in red, the user needs to finish the configuration of this particular asset.

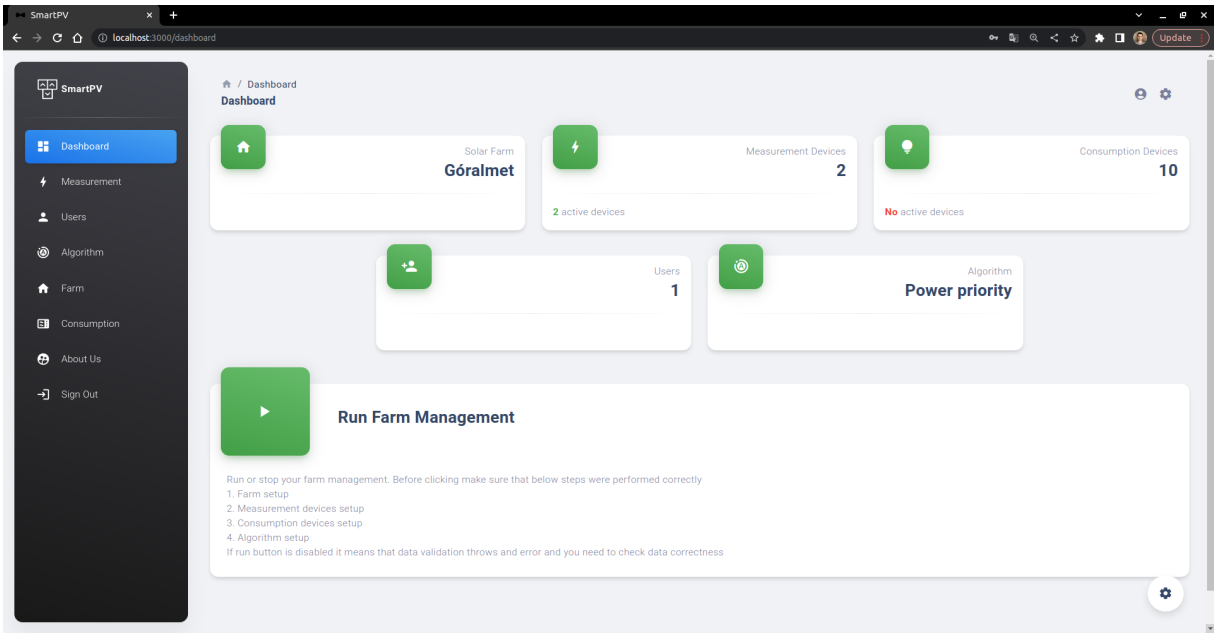

Figure 25: Dashboard - web application

When all of the necessary information is filled out, we can finally set our system up and running by clicking the green icon in the **Run Farm Management** section of the **Dashboard** view.

And... That's it! The system is working as configured by the admin. Any changes can still be made even during the run-time - we can still add, edit and remove devices, manage system's users and change the ongoing algorithm. It is all up to the user.

5.2.9. About us

In the "About us" section we included general information about ourselves as creators of the system along with links to our respective Github profiles.

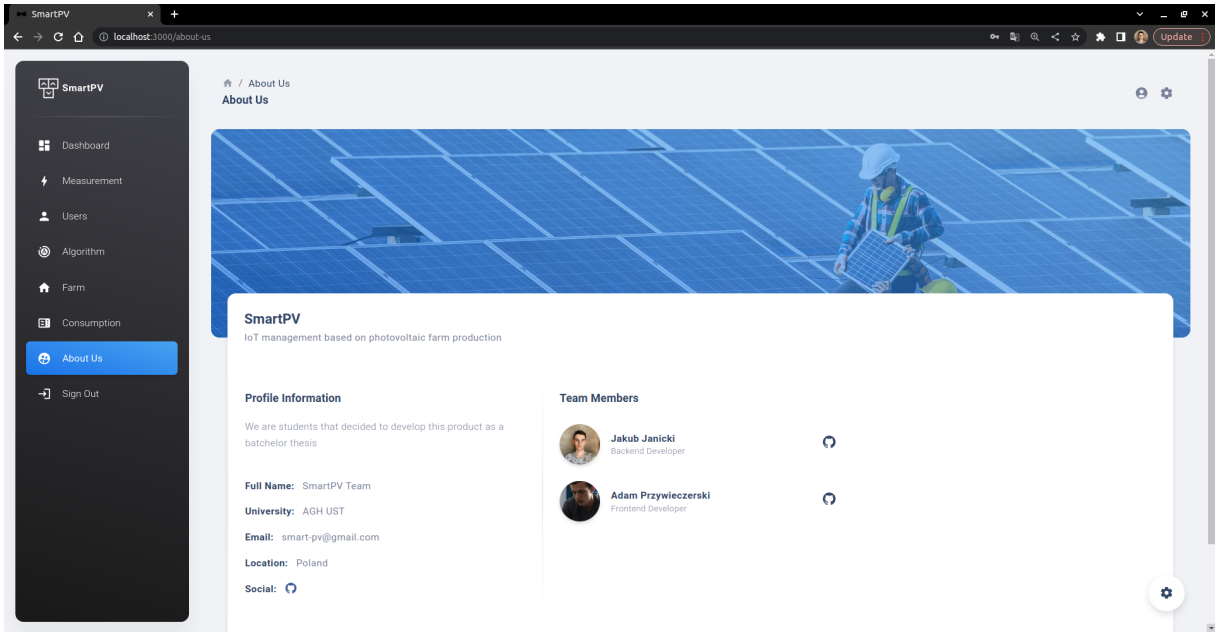

Figure 26: About us - web application

5.2.10. Log in to the mobile application

To log in to the mobile application we need to have a previously registered account in the system, which can only be assigned by an admin. If we already have one, we just need to pass the credentials when prompted to.

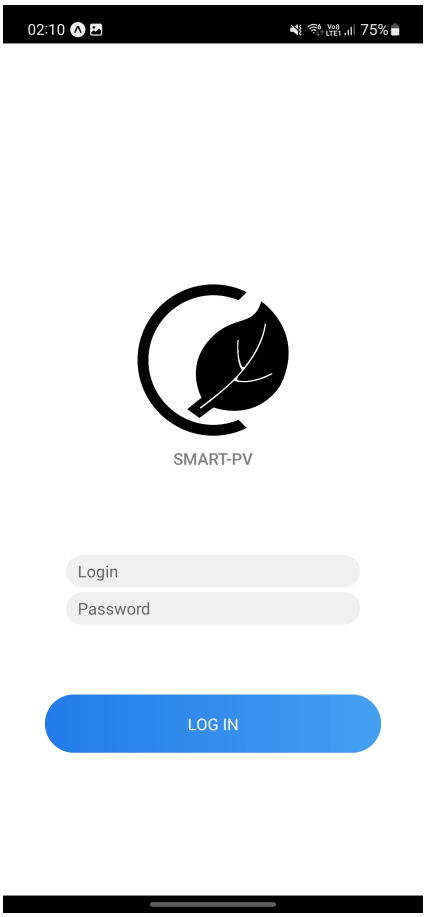

Figure 27: Log in - mobile application

5.2.11. General information screen

After logging in we are taken to the home screen of the application, where we can see basic information about the farm, running devices, and a graph with summed readings from the farm.

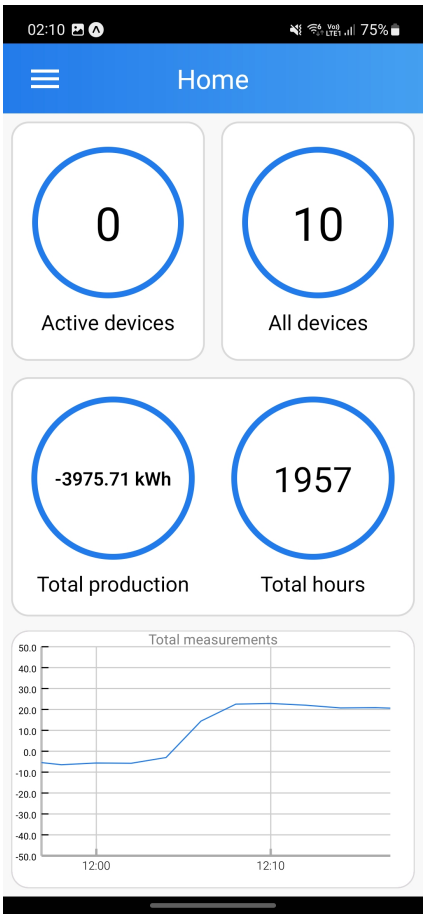

Figure 28: Home screen - mobile application

To navigate to other tabs, we can use the side menu accessible by clicking the icon in the upper-left corner or by swiping right from the left edge of the screen.

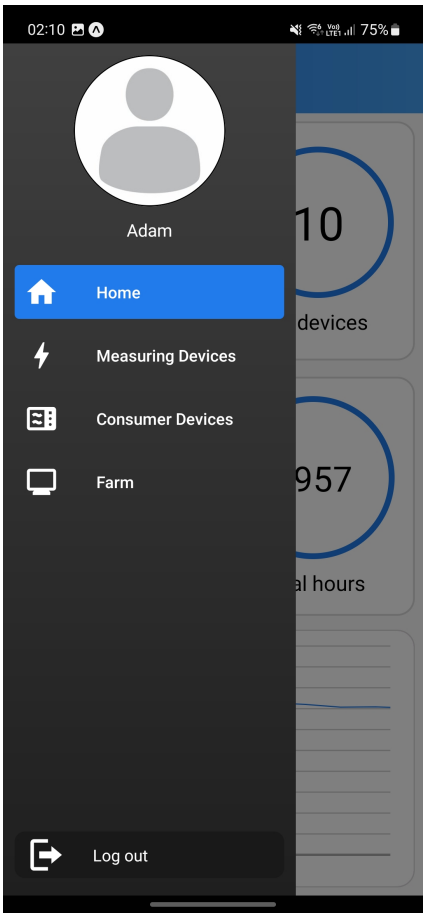

Figure 29: Side menu - mobile application

5.2.12. Devices overview

Both measuring and consumer devices are presented in a similar way. We have separate lists for each of them filled with their miniatures, which hold basic information about each of them.

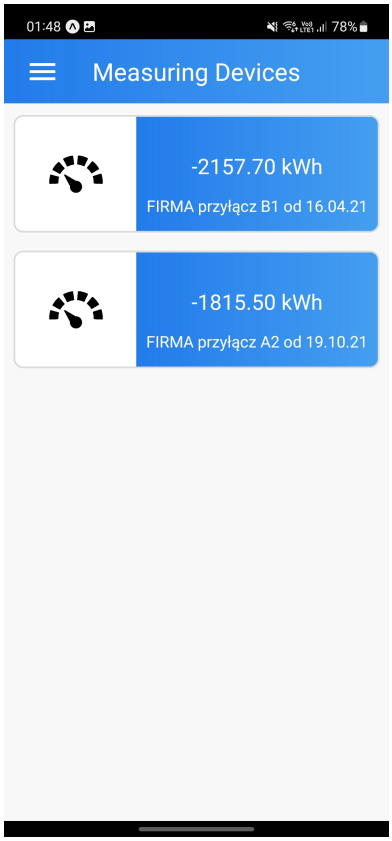

Figure 30: Measuring devices - mobile application

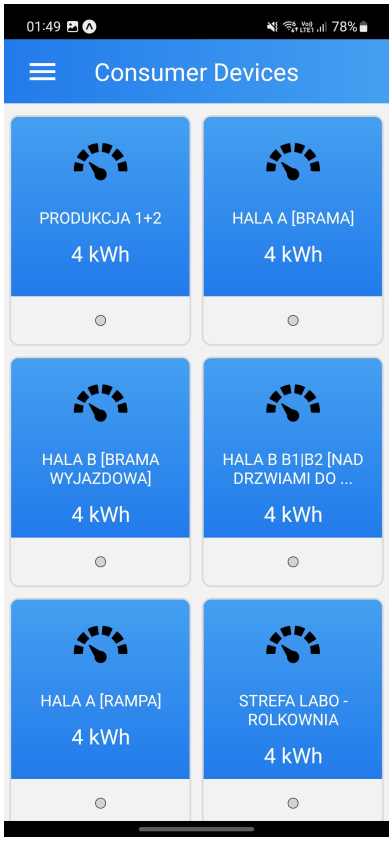

Figure 31: Consumer devices - mobile application

We can access each device for more specific information by clicking on its miniature.

5.2.13. Measuring device

On this screen we have access to data specific to the selected measuring device. At the top we can see the graph of measured energy produced by the photovoltaic installation with the option of changing the range of the showcased readings. Below, we have information about the device: its name, the farm that it belongs to, and its latest measurement.

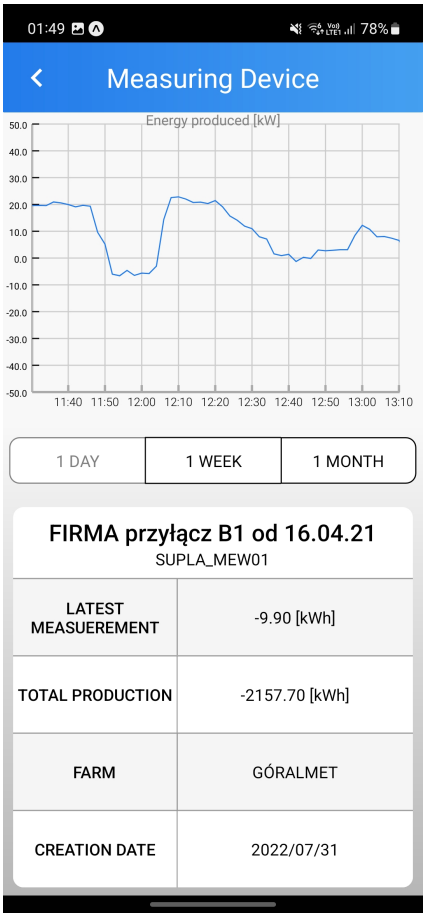

Figure 32: Measuring device - mobile application

5.2.14. Consumer device

The consumer device screen is similar to the measuring device screen, although they differ in the data they present. The graph at the top shows the activity chart of the device over the selected period of time. In the data section of the screen we see information such as: name of the device, current status, performed task and power consumption.

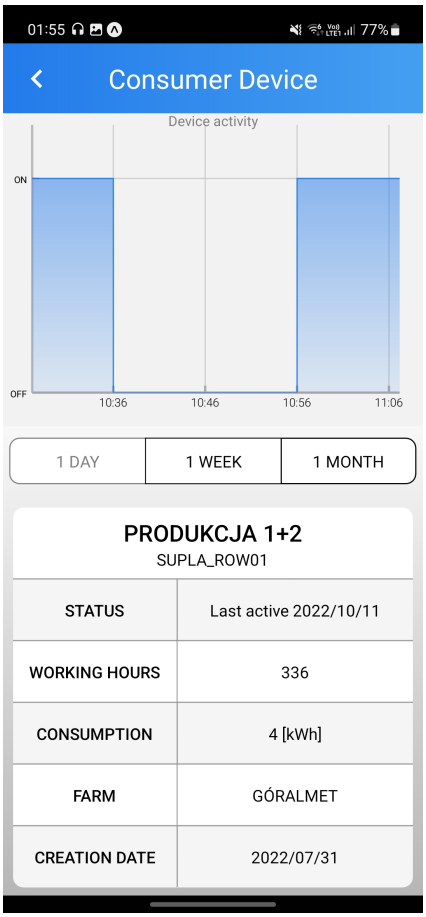

Figure 33: Consumer device - mobile application

5.2.15. Farm

The farm screen offers general information about the farm and the values of parameters based on which it is operating.

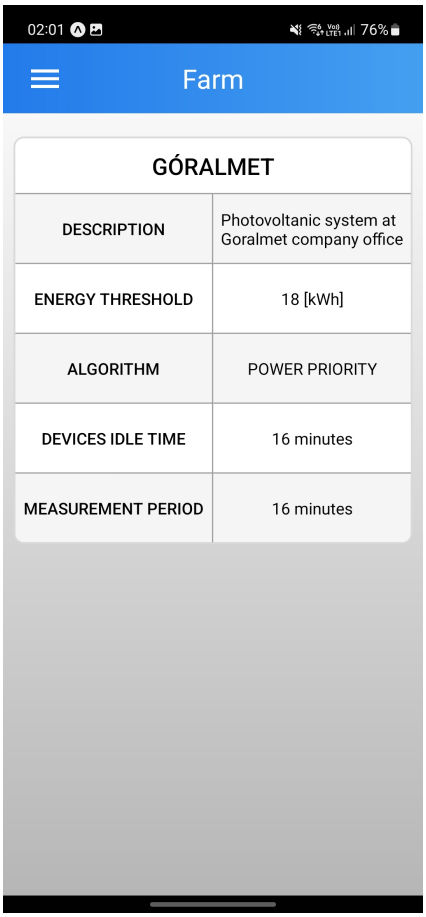

Figure 34: Farm - mobile application

5.3. System testing

We had the opportunity to test our system in a real-life setting in a warehouse dedicated to the storage of metal products such as chains and metal beams. During the summer, there is an increased need for cooling of the interior to keep the temperature at a constant level.

5.3.1. Site specification

The object is fitted with 2 clusters of solar panels - one on the side of the building and one on the roof. Inside there are 10 AC units, each with a power demand of 9kW. Their efficiency is enhanced by the presence of air mixers, which allow for an even distribution of heat along the vertical axis. The maximum power of the panels is 115 kW, the inverters 108 kW, and the electrical connection to the outside network can hold 2x40 kW.

Figure 35: Solar panels - side view

Figure 36: Solar panels - roof view

To measure the difference in the energy fed into the central grid and the energy used by appliances, there are 2 measuring devices (Zamel Supla Mew-01) installed - one for each electrical connection. Communication with the air conditioners is performed with the use of controlling devices (also produced by Zamel - Supla Row-01).

5.4. Results

On the site, we managed to run one algorithm, the interval algorithm, with only the basic capabilities of the system. The customer did not define additional functionalities, so the procedure

was only responsible for turning the devices on and off at appropriate times. Nevertheless, during the development process, we tried to gather as much data from the site as we could. Because of that, we were able to reproduce a real environment to test our system in. In the sections below we will present the results of work of the Interval algorithm running on the location and the Priority algorithm simulated on the gathered data.

5.4.1. Priority algorithm

We ran the simulation using this algorithm for specific days. The results are as follows.

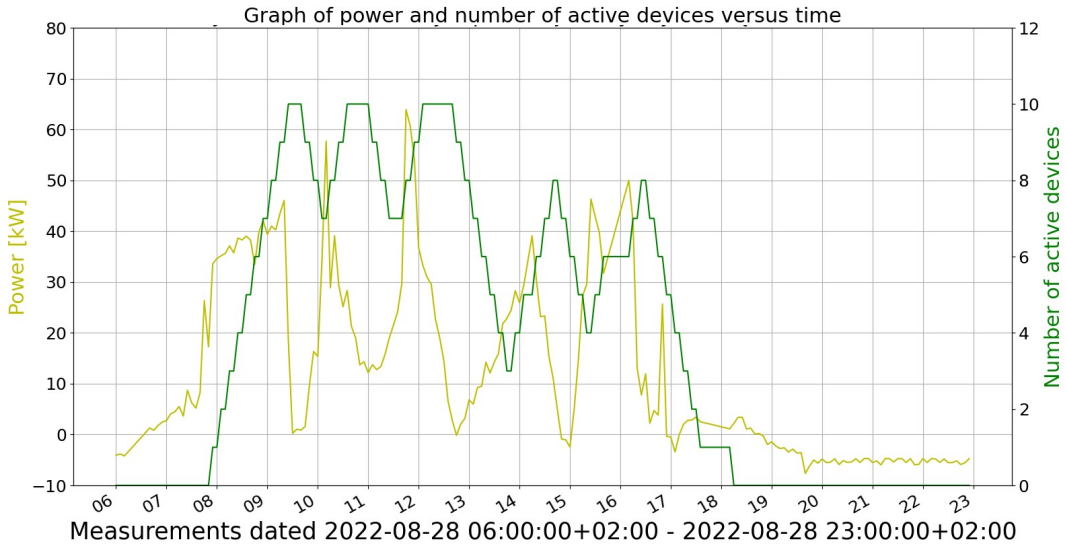

Figure 37: Priority algorithm - simulation 1

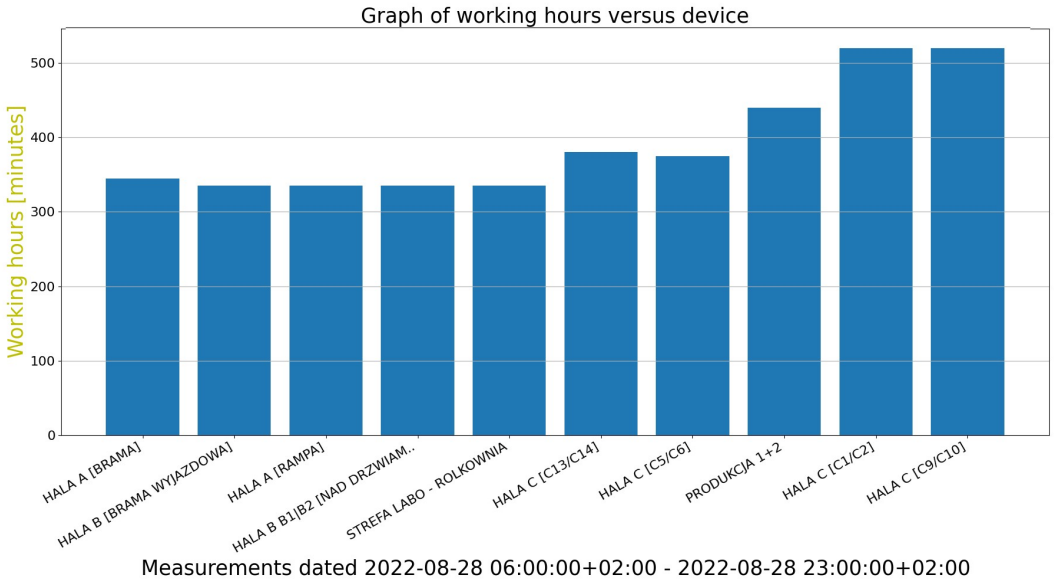

Figure 38: Priority algorithm - device activity

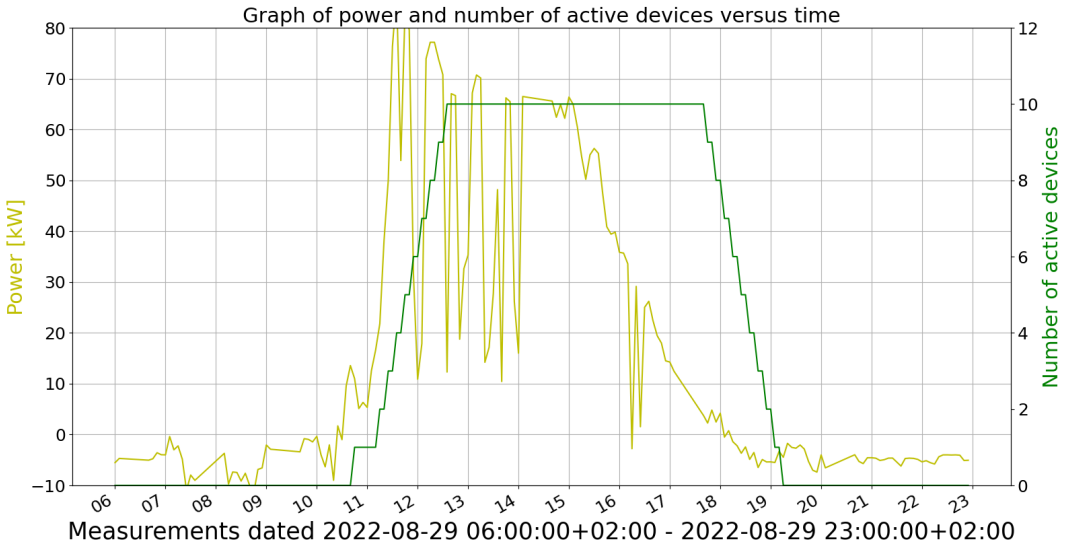

Figure 39: Priority algorithm - simulation 2

5.4.2. Interval algorithm

The interval algorithm worked throughout the summer in the real location, allowing for data gathering and confirmation of the working state of the system.

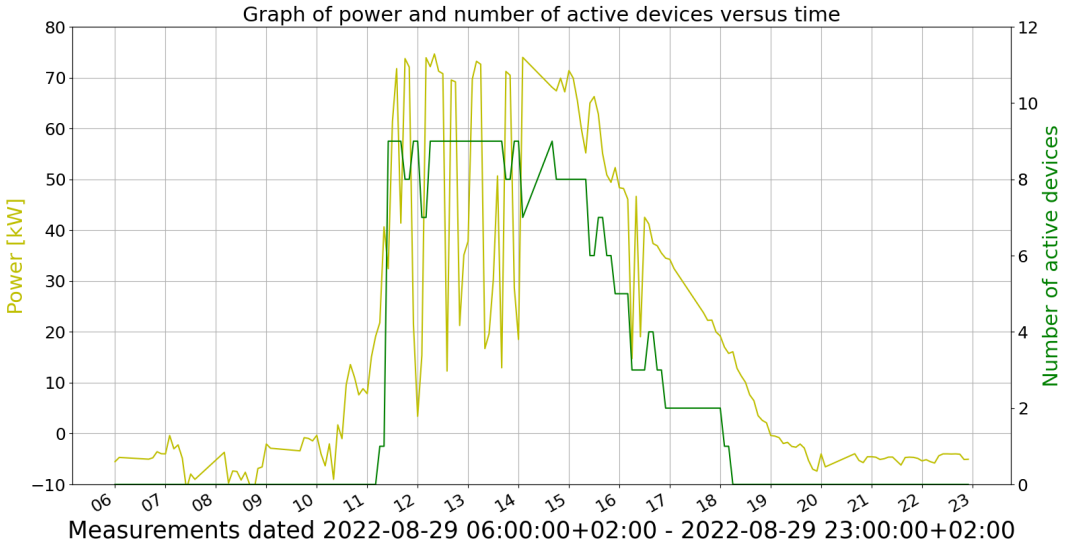

Figure 40: Interval algorithm - system behavior 1

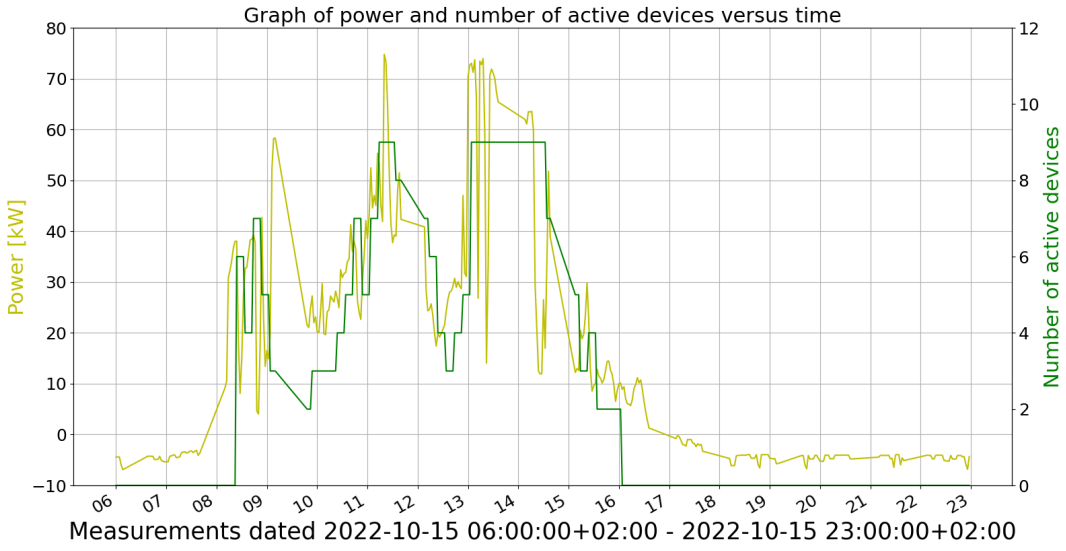

Figure 41: Interval algorithm - system behavior 2

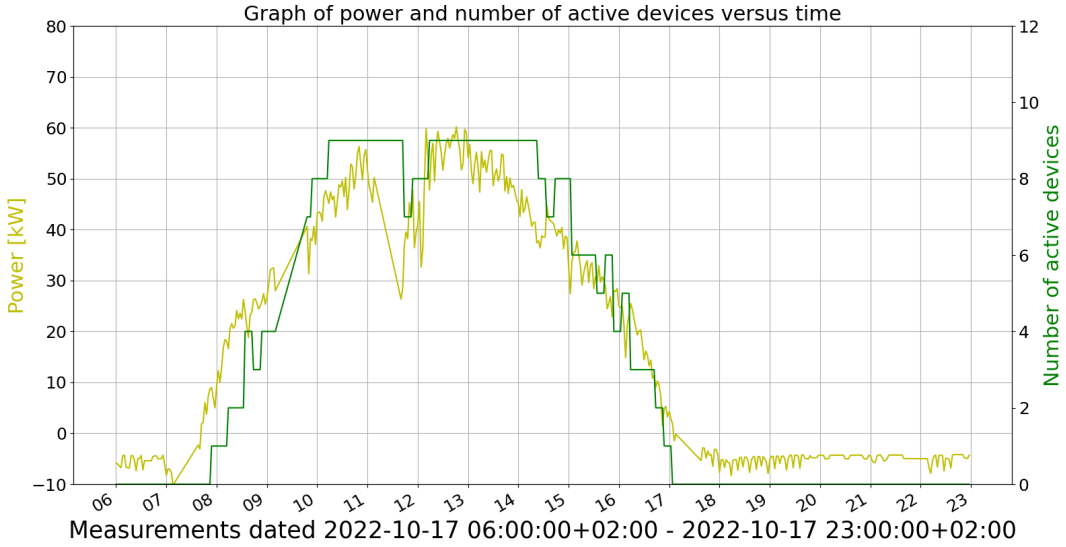

Figure 42: Interval algorithm - system behavior 3

5.4.3. Charts analysis

We tried to include all kinds of charts, from constantly sunny days to rainy and cloudy days, where energy production might have been inconsistent. The latter provide the most interesting problems in terms of our system's working.

On the charts we can see the differences in how our 2 main algorithms act. The Priority variant's reaction to a sudden influx of energy is incremental - in the algorithm's iteration, we only

approve a single action on one device. In contrast, the Interval variant allows for a faster reaction time as many actions can be defined in a single interval.

We can also see the latency in the system's actions despite the conditions. It is caused by the way our algorithms make decisions, they work on estimates based on previous measurements. Another reason is the minimum energy threshold available for each farm, which must be achieved to perform any additional tasks.

Because of the use of estimates in the system's decision-making process, the algorithms are fairly resistant to short-term energy fluctuations, they do not make decisions at first sight of changing conditions, knowing that they might be temporary - e.g. a cloud passing by.

On the time graph (Fig. 35) we can also see that for the Priority algorithm, some appliances worked significantly longer than the others. It simply means that because of their higher priority they were favored by the system.

5.4.4. Statistics

The gathered data comes from an extended period of time, when our system was working in the warehouse - from August 21st to December 1st. During those months, we collected some additional information about the overall state of the whole network.

The total energy transferred to the central grid amounted to 10100 kWh. The total work time of all devices combined reached 3450 hours.

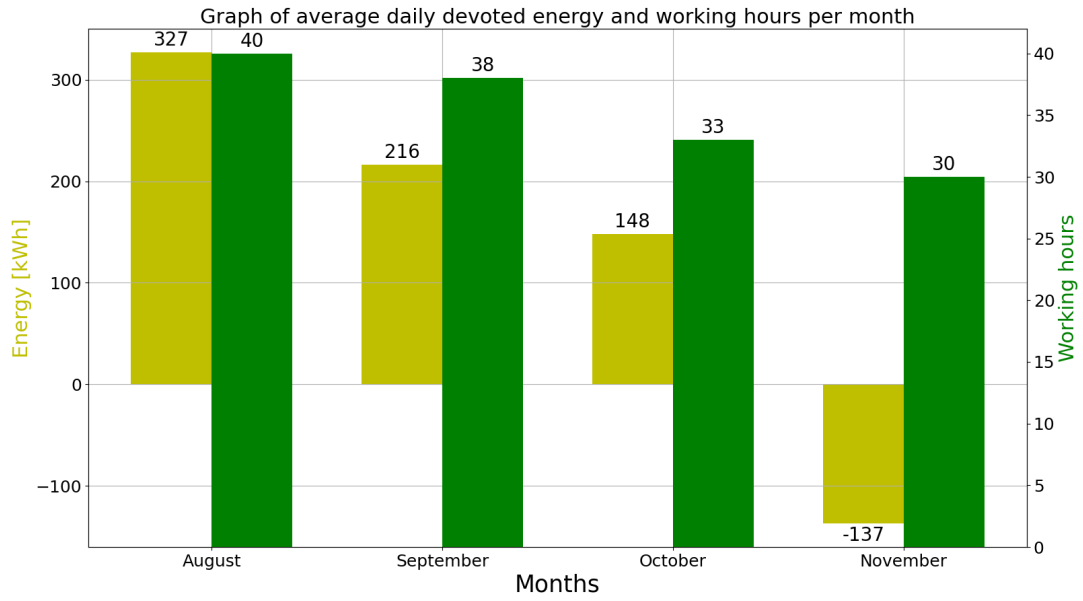

Figure 43: Average daily devoted energy and working hours per month statistics

While analyzing the data, we have to keep in mind the it is not fully complete. Because of outages in the system, problems in communication or device malfunctions, breaks in data gathering occured.

5.5. Code and installation process

We have created a GitHub organization for our project which consists of all code that we had written. Each sub-product has its own repository with a readme file where the installation process is included. The server and all necessary microservices as well as the web application are deployed with Heroku. The mobile application was packed into a .apk file. Not only links to products are shared but also test user credentials. The test farm was created with all of its data mocked so it can be used for presentation purposes. Link to our smart-pv-team organization: *https://github.com/smart-pv-team*

5.6. Development ideas

Having finished the first version of our product, we know the challenges of its development process and we are aware of its most difficult functionalities to implement and use. Over the past couple of months, we also came up with new feature ideas that we were unable to fit into this version of the product. Here are the ways in which we would like to improve our system in the future.

5.6.1. Addition of a Schedule

At the moment, our system is trying to use all of its associated devices efficiently. If there is not enough energy to power any of them, the system will not act. Each device has its own metric, on the basis of which it may be selected to perform a task. However, some customers may not value the efficiency of the system, but its ability to communicate with IoT appliances. In such a case, we would like to provide the users with the ability to create a schedule for all devices. On the basis of this schedule, every device would perform given tasks at a certain time of the day, week, or month.

5.6.2. Measuring the energy produced by a farm

We would like to find a way to measure the exact amount of energy produced by the farm. We base our system's working on the amount of energy being given away to the central grid because neither the suppliers of energy meters nor inverters provide an API for a more detailed view. In the case of the Interval algorithm we may not know exactly the power consumption of performing a particular task, which forces us to use estimated values for energy usage and hence results in an estimated energy production.

5.6.3. Extending the communication methods

Some IoT devices use different methods of communication instead of the HTTP protocol that we assumed. In the future we could extend the system capabilities to also include various types of communication protocols. The system's structure would not be greatly affected as most of the implementation could be done in the adapters section of the system.

5.7. Summary

Our product turned out as we hoped, and both client parties were happy with its final version. The system works as intended, allowing for efficient use of the surplus of energy generated by the local solar installation. Having the ability to test the system in a real life setting was

incredibly helpful in making sure that the whole infrastructure works correctly and also made catching potential bugs a lot easier.

Despite many difficulties along the way, we managed to successfully finish the development of the system in its first version. There is definitely still room for improvement regarding the performance of the entire product, as there are also countless ways to extend its capabilities with new features. In the future, if given enough free time, we can definitely see ourselves coming back to this project and continuing the work on the system to make it even better.

References

[1] Dependency injection. https://en.wikipedia.org/wiki/Dependency_injection.

[2] Dependency inversion. https://en.wikipedia.org/wiki/Dependency_inversion_principle.

[3] Domain-Driven Design. https://en.wikipedia.org/wiki/Domain-driven_design.

[4] Github Actions. https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects.

[5] Hexagonal architecture - Wikipedia. https://en.wikipedia.org/wiki/Hexagonal_architecture_(software).

[6] Lombok. https://projectlombok.org/api/.

[7] React Hook Form. https://react-hook-form.com/.

[8] React-Native-Responsive-Linechart. https://www.npmjs.com/package/react-native-responsive-linechart.

[9] React Redux. https://react-redux.js.org/.

[10] Waterfall Model. https://en.wikipedia.org/wiki/Waterfall_model.

[11] Material Dashboard 2 React Template. https://www.creative-tim.com/product/material-dashboard-react, 2022.

[12] React Documentation. https://reactjs.org/docs/react-api.htmlreactpurecomponent, 2022.

[13] Scipy Library Documentation. https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.simpson. 2022.

[14] dr inż. Jacek Dajda. Lectures of Iżynieria Oprogramowania subject AGH. 2022.

[15] Główny Urząd Statystyczny. Energia ze źródeł odnawialnych w 2020 roku. https://stat.gov.pl/en/topics/environment-energy/energy/energy-from-renewable-sources-in-2020,3,13.html.

[16] V. Halili. Hexagonal architecture: What is it and why should you use it? https://cardoai.com/what-is-hexagonal-architecture-should-you-use-it/, 2022.

[17] Hgraca. DDD, Hexagonal, Onion, Clean, CQRS, ... How I put it all together. https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/, 2017.

[18] B. Knutel, A. Pierzyńska, M. Dębowski, P. Bukowski, and A. Dyjakon. Assessment of energy storage from photovoltaic installations in poland using batteries or hydrogen. *Energies*, 13(15), 2020.

[19] R. Marks-Bielska, S. Bielski, K. Pik, and K. Kurowska. The importance of renewable energy sources in poland's energy mix. *Energies*, 13(18), 2020.

[20] R. C. Martin. A Craftsman's Guide to Software Structure and Design. 2017.

[21] mgr inż. Konieczny Marek. Lectures of Architektura Rozwiązań Chmurowych subject AGH. 2022.

[22] Ministerstwo Klimatu i Środowiska. Energy Policy of Poland until 2040 (EPP2040). https://www.gov.pl/web/climate/energy-policy-of-poland-until-2040-epp2040.

[23] Ministerstwo Klimatu i Środowiska. Mój prąd. https://mojprad.gov.pl/component/content/article?id=23Itemid=605.

[24] Ministerstwo Klimatu i Środowiska. Nowy system rozliczania, tzw. net-billing. http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/.

[25] P. Olczak. Energy productivity of microinverter photovoltaic microinstallation: Comparison of simulation and measured results; poland case study. *Energies*, 15(20), 2022.

[26] J. Rosak-Szyrocka and J. Żywiołek. Qualitative analysis of household energy awareness in poland. *Energies*, 15(6), 2022.

[27] M. Trela and A. Dubel. Net-metering vs. net-billing from the investors perspectivem-dash;impacts of changes in res financing in poland on the profitability of a joint photovoltaic panels and heat pump system. *Energies*, 15(1), 2022.

[28] J. Woźniak, Z. Krysa, and M. Dudek. Concept of government-subsidized energy prices for a group of individual consumers in poland as a means to reduce smog. *Energy Policy*, 144:111620, 2020.